(1)

EXPLANATION GENERATION

IN

EXPERT SYSTEMS

(A Literature Review And Implementation)

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

Captain VANCE MCMILLAN SAUNDERS
B.S., Wright State University, 1984

1989
Wright State University

WRIGHT STATE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

November 1989

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Captain Vance McMillan Saunders ENTITLED Explanation Generation In Expert Systems (A Literature Review And Implementation) BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

Thesis Director

Department Chair

Committee on
Final Examination

Robert C Hock

Dean of the School of Graduate Studies

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 89-142
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT WRIGHT STATE UNIV 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION AFIT/CIA
6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6583
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) (UNCLASSIFIED)

EXPLANATION GENERATION IN EXPERT SYSTEMS (A LITERATURE REVIEW AND IMPLEMENTATION)

12. PERSONAL AUTHOR(S)
VANCE MCMILLAN SAUNDERS

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
THESIS/~~DISSERTATION~~xxx	FROM _____ TO _____	1989	145

16. SUPPLEMENTARY NOTATION
APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF	22b. TELEPHONE (Include Area Code) (513) 255-2259 22c. OFFICE SYMBOL AFIT/CI

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"

ABSTRACT

Captain Saunders, Vance McMillan. M.S., Department of Computer Science and Engineering, Wright State University, 1989. Explanation Generation in Expert Systems (A Literature Review and Implementation)

Today's technology provides tremendous amounts of information at incredible speeds. In order to make this information useful for more complex, significant problem solving applications, intelligent computer software systems are needed. The Expert System (ES) technology of Artificial Intelligence (AI) is one solution that is emerging to meet this need. However, as this technology continues to develop and as we begin to use expert machines more and more, it is crucial that we demand the same explanatory capability from these mechanical experts as we do from human experts. The credibility of human expertise is established by an expert's ability to explain his expertise. The credibility of mechanical expertise must be established in the same way. Additionally, ESs are very complex, so[illegible]cated systems. In order to verify their accuracy and correctness, ESs must be able to explain what they are doing and why.

This thesis examines the Explanation Facilities (EFs) of ESs by first conducting an extensive literature review of this topic, and second, by implementing an EF for a frame based ES shell. The purpose of the literature review is to gain a general understanding of EF research and development while the purpose of the implementation effort is to investigate the specifics of: explaining a frame-based ES, adding an EF to an existing ES, and using Ada as the implementation language for this AI application.

Contents

Chapter 1

Introduction

1.1 Background

The purpose of this thesis is to examine the *explanation generation capabilities* of expert systems. This is based on the need to develop *intelligent* software systems in order to **effectively** process and use the tremendous amounts of data that technology continues to provide and to develop these systems using sound software engineering practices and principles. One technology that is being designed to provide these *intelligent* software systems is the Expert System (ES) technology of Artificial Intelligence (AI). However, as the technology of building *computerized experts* transitions more and more from the academic laboratory to different operational, on-line environments (business, industry, defense, etc.), the importance of incorporating the above mentioned software engineering practices into the development of these ESs needs to be re-emphasized. Due to the complexity and seriousness of the problems ESs are being developed to solve, **we simply can't afford the cost associated with producing systems that are unreliable or produce incorrect results.**

This thesis examines one capability that provides significant contributions towards establishing the credibility and reliability of ESs. This is the capability of ESs to **explain** their internal processing mechanisms and reasoning processes to a human user. Interestingly, we require human experts to possess this capability in

order for us to accept their expertise as legitimate. It is even more important that we require these same capabilities from computerized (*mechanical*) experts.

This examination of explanation generation in ESs will be conducted in two parts. The first part conducts a relatively intensive review of the available literature on this subject in order to obtain a basic understanding of what is being done in the area of explanation generation research and development. This information is then used in the second part of this thesis to design and implement an **E**xplanation **F**acility for a **F**rame-based **E**xpert **S**ystem **S**hell (EFFESS) using the software engineering intensive *Ada* programming language. (This work is part of a larger *Ada and AI* research and development effort currently being conducted at Wright State University.)

1.2 Literature Review

The literature review is presented in the next seven chapters of this thesis. It begins in Chapter 2 by precisely defining what is meant by *explanation generation* in ESs, discussing several specific reasons why EFs are important requirements for ESs, and by describing a Functional Framework in which to analyze the EFs of ESs. All of this is presented for the purpose of establishing a common ground of understanding to be used throughout the rest of the thesis.

Chapters 3 — 8 present examinations and discussions of several different efforts in explanation generation research and development. Chapter 3 discusses the work done by Edward Shortliffe and Bruce Buchanan on the MYCIN expert system. This work is recognized as the first in EF research and development and therefore provides the logical place to begin the literature review. Chapter 4 examines the work done on three ESs that are direct descendants of MYCIN. These systems: TEIRESIAS, GUIDON, and NEOMYCIN represent different efforts to increase the effectiveness

of MYCIN's original EF. Chapter 5 looks at the ongoing work of William Swartout and involves three different ESs. *The Digitalis Therapy Advisor With Explanations*, the first of these systems, was concerned primarily with overcoming some of the limitations identified in the original MYCIN EF. However, the two systems that followed this initial work, XPLAIN and EES, provide significant new approaches in EF development. Chapter 6 discusses the work being done by B. Chandrasekaran and deals with the topic of *generic tasks* and the contributions they make to ES development and the generation of explanations. Chapter 7 looks at three independent efforts in EF research and development. The first is the work of J. L. Weiner in BLAH. The focus of this effort is in developing ways to restructure explanations in order to reduce their complexity and therefore make them easier to understand. The second effort is the work done by Robert Rubinoff in CLEAR. The primary focus of Rubinoff's work is in producing explanations of terms and concepts that aren't understood by the user of the system. The third effort is the work of Michael Wick and James Slagle in JOE. Of interest here is their use of journalistic principles in generating explanations. Finally, Chapter 8 discusses explanation in the context of Roger Schank's work on *understanding*. This chapter briefly re-examines each of the previous EF efforts, and, for the purpose of review, evaluates each of them in the context of Schank's *understanding spectrum*. It then looks at some possible directions for future EF research and development.

1.3 Implementation

The purpose of the implementation is to provide hands on experience in designing and developing an EF that is developed in Ada and is **added** to an existing, frame-based ES shell. Three areas of interest are presented in discussing this project: its scope, the ES shell it uses, and its design and functionality. Chapter 9 defines the

scope of EFFESS by identifying its initial requirements and by using these requirements to filter through the literature review in order to identify those concepts and ideas applicable to its implementation. Chapter 10 presents a detailed examination of the **frame-based** *knowledge representation structure* (KRS) used in the ES shell. As this is the primary object or structure that is to be explained, understanding it precisely is important. Chapter 11 then discusses the specific design of EFFESS and descibes each of the functions included in it. This chapter also analyzes EFFESS with respect to the Functional Framework presented in Chapter 2. These three chapters provide the basic examination of this EF implementation effort. However, one final chapter is provided in order to present some concluding observations concerning the EFFESS project and to identify some areas for future research.

Chapter 2

Establishing A Common Ground Of Understanding

2.1 Overview

The purpose of this chapter is to establish a common ground of understanding upon which to conduct a detailed examination of EFs (also referred to as *explanation generation*) in expert systems. This will be done by first defining precisely what we mean by the word *explanation*, second by discussing several major factors that establish and emphasize the importance of EFs, and finally by looking at the three major functions that must be considered when developing an EF.

2.2 "Explanation" Defined

Before we can begin to examine the work that has and is being done in the area of expert system EFs it is important that we understand exactly what we mean by *explanation*. The Random House College Dictionary defines *explanation* as:

> "to make plain, clear, or intelligible something that is not known or understood". [33]

While this defines the basic meaning of the word and therefore provides the basic purpose of an EF, Chandrasekaran et al. [8] identifies two different types of explanation that this general definition encompasses with respect to ESs. These two types of

explanation are *explaining the world* and *explaining decisions*. *Explaining the world* refers to the explaining of a data set or the providing of logical reconstructions to show how certain theories are explanations of some observed phenomena. Expert systems that provide these types of explanation are concerned with the objects and processes that are **external** to their own processing mechanisms. Examples given were diagnostic type ESs such as DENDRAL [15], INTERNIST [25], and RED [20]. *Explaining decisions* on the other hand, refers to the explaining of one's own decisions. Expert systems that provide this type of explanation are concerned with the objects and processes that are **internal** to their own processing mechanisms. There are many ESs that provide varying degrees of this type of explanation also. Several of these ESs will be identified and discussed in subsequent chapters of this paper as it is this internal type of explanation that we are interested in as we discuss the Explanation Facilities of ESs. Therefore, in concurrence with Clancy, Hasling, and Rennels, we define *explanation* to mean:

> "the ability of a program to discuss what it is doing in some understandable way". [36, page 3]

An EF then, is the mechanism an ES uses to provide this explanation.

2.3 The Importance Of Explanation Facilities

The first step in establishing the importance of EFs is to recognize that,

> "Effective methods of generating explanations of both system queries and conclusions have long been considered an important feature of expert systems." [11, page xiv]

While some of the first ESs did not have EFs, it didn't take long for their need to be identified. As the domain of problem-solving applications to which ESs were being applied continued to grow in complexity and as the costliness of making errors in these problem-solving processes continued to rise, so did the importance of EFs. Today, EFs are considered as important to an ES as its knowledge base, control

strategy, or inferencing mechanism. In fact, Firebaugh [17], while identifying ESs as belonging to a more general class of computer systems known as *knowledge-based systems*, emphasizes the fact that it is **the explanation facility of an ES** that distinguishes it from other knowledge-based AI programs. In other words, **an expert system without an explanation facility is not an expert system.** Regardless of whether one agrees with Firebaugh or not, few individuals (if any) working in ES research and development will argue against the need for EFs. There are four major reasons why this is true. Three of these reasons are explicitly stated in much of the literature on explanation. The fourth is implicitly derived from the fundamental goal of AI. We'll begin by discussing this reason first, followed by the other three.

In his book, ARTIFICIAL INTELLIGENCE A Knowledge-Based Approach, Morris W. Firebaugh states,

> "The declared goal of artificial intelligence research is to teach machines to "think", that is, to display those characteristics usually associated with human intelligence." [17, page 3]

If we extrapolate on this idea with respect to ESs then the fundamental goal of an ES is to **think** like a human expert; to display those characteristics usually associated with a human expert. Accepting this to be true, Swartout and Smoliar [32] identify four basic characteristics of a human expert that we can apply to ESs. Paraphrased, these four characteristics of an expert are:

1. The ability to solve problems within his domain of expertise.

2. The ability to analyze a problem statement and determine whether or not he is capable or qualified to solve the problem (whether or not it is within his realm of expertise).

3. The ability to **explain** his problem-solving behavior.

1. The ability to adapt his problem-solving strategies to new and unusual situations.

If we are to achieve our fundamental goal of building ESs that display the characteristics of a human expert and if one of the characteristics of a human expert is the ability to explain his problem solving process, then ESs must also display this characteristic.

The second reason EFs are important stems from the rationale requiring an expert to be able to explain himself. Schank, in his work on explanation [27] points out that we humans do not allow other human beings to come up with new ideas, concepts, etc. unless they can explain themselves. He goes on to say that while we will accept unusual or unexplained demonstrations of brilliance for a short while, eventually we will begin to question and object, thinking that somehow we are being fooled. Eventually we no longer accept the demonstrated brilliance as being legitimate. Thus, experts must be able to explain themselves because we humans require it in order to have confidence in what they are telling us. EFs satisfy this same requirement for ESs. Using the EF, human experts in the application domain of a particular ES can determine if the system is performing as they expect it to, thereby assuring themselves of its *credibility*.

While reason number two argues that we humans **won't** accept ESs that lack an explanation facility, reason number three argues that we **can't** accept ESs that lack an explanation facility. Says Forsyth,

> "The explanation facility should not be regarded as an optional extra. Donald Michie (1982) and others have warned about the dire consequences of systems which do not operate within the 'human cognitive window', i.e. whose actions are opaque and inexplicable.
>
> If we are to avoid a succession of Three-Mile- Island-type disasters or worse, then our expert systems must be open to interrogation and inspection. In short, a reasoning method that cannot be explained to a person is unsatisfactory, **even if it performs better than a human expert**." [18, page 14]

The problems associated with the development of erroneous, unreliable, and unmaintainable software systems is one of the most serious problems in the field of computer science. The entire Software Engineering discipline is dedicated to solving this problem. ESs that do not have a capability for *interrogation and inspection* will only aggravate and complicate the problems Software Engineering is trying to solve. This is definitely an unacceptable situation. Thus, the third reason EFs are important is because they provide an invaluable debugging tool, allowing knowledge engineers and system designers to examine the ES for accuracy and correctness.

The fourth reason EFs are important can almost be viewed as a by-product of the previous two capabilities. If an EF can explain its reasoning process to an expert or its control structure and data relationships to a knowledge engineer, then it stands to reason that a novice user with little knowledge of the application domain could use this same information as a learning/teaching aide. This is, in fact, the case, and, as we shall see, much has been done with regards to explanation generation and tutorial ES applications.

These then are the major reasons EFs are considered to be such important components of ESs. Let's now look at the three functions that must be considered in order to produce them.

2.4 Functional Framework For Producing Explanations

The literature identifies three major functions that must be considered in order to generate or produce explanations in an ES. These three high-level functions constitute the basic framework in which all work on explanation generation is currently being done. Interestingly, these three functions (this functional framework) are identified in the literature in several different ways. Clancy et al. [36] identifies

them as *epistemologic issues*, *user modeling*, and *rhetoric*. Chandrasekaran et al. [8] identifies them as *generating explanations basic content*, *responsiveness*, and *the human-computer interface*. Still others [14] do not provide names or titles for these functions at all. For ease of reference, we will refer to them as Functions 1, 2, and 3 in our discussion. However, the names used to identify these functions is not the important issue here but rather the understanding of their contents and applications. A high-level description of each will now be presented.

Function 1: Remembering our established definition of explanation, the content of any explanation to be generated is based on the internal examination (introspection) of its own problem-solving mechanism or behavior. Function 1 concerns itself with identifying ways to model the contents of this problem-solving mechanism (the knowledge and reasoning process of the system). This is the foundation of the entire explanation process because this is what has to be and is going to be explained. This is the primary information the user wants to know about and the developer wants to validate and check. Therefore, the knowledge and reasoning process must be represented in well defined, well structured methods or formalisms. In addition, these methods or formalisms must be appropriate for the specific problem-solving task being addressed and must be able to be examined by the system. However, attacking this modeling process from the *knowledge and reasoning process* level of an ES is to approach it from a very high-level, abstract point of view and is difficult to do. By identifying three different types of explanation that can be produced from the *knowledge and reasoning process* of an ES, Chandrasekaran et al. [8,9] have provided a more detailed level of abstraction from which to attack this problem. These three types of explanation are now described.

1. **Type 1** explanations are concerned with explaining why certain decisions were or were not made during the execution (runtime) of the system. These

explanations use information about the relationships that exist between pieces of data and the knowledge (sets of rules for example) available for making specific decisions or choices based on this data. For example, Rule X fired because Data Y was found to be true.

2. **Type 2** explanations are concerned with explaining the knowledge base elements themselves. In order to do this, explanations of this type must look at *knowledge about knowledge*. For example, knowledge may exist about a rule that identifies this rule (this piece of knowledge) as being applicable ninety percent of the time. A type 2 explanation could use this information (this knowledge about knowledge) to justify the use of this rule. Other knowledge used in providing this type of explanation consists of knowledge that is used to develop the ES but which does not effect the operation of the system. This type of knowledge is referred to as **deep** knowledge.

3. **Type 3** explanations are concerned with explaining the runtime control strategy used to solve a particular problem. For example, explaining why one particular rule (or set of rules) was **fired** before some other rule is an explanation about the control strategy of the system. Explaining why a certain question (or type of question) was asked of the user in lieu of some other logical or related choice is another example. Therefore, type 3 explanations are concerned with explaining how and why the system uses its knowledge the way it does, a task that also requires the use of **deep** knowledge in many cases.

These three types of explanation help divide Function 1 concerns into more specific, refined sub-areas.

Function 2: This function concerns itself with providing an explanation to the user based on that user's particulars needs and abilities. The idea here is that every user is different. Each has a different level of understanding about the problem

domain. Each has a different reason for wanting a particular explanation. Based on these differences, it may not be necessary for all available information about an explanation to be provided. One way that this is accomplished is by creating a *model of the user* (a database of knowledge that the user possesses) and then using this model as a gage for determining the degree of explanation to present to that particular user.

Function 3: This function concerns itself with **how** to convey or present the information to the user. Should natural language be used or will source code statements suffice? What about graphical displays? Should text and graphics be combined? These are the types of questions (or concerns) this function must consider.

This section has described a functional framework for generating explanations. The question of **how** these functions are being achieved is addressed in subsequent chapters of this thesis. Most of what will be discussed, however, pertains to Function 1. The reason for this is two-fold. As already mentioned, the concerns of this function constitute the foundation of any explanation. No matter how well the user's needs and abilities are understood (Function 2) and no matter how appropriately the explanation can be displayed (Function 3), if the information content of the explanation is bad, the explanation will be bad. Furthermore, from a **purest** viewpoint of explanation, Function 1 deals with the **real** concerns and issues. Determining how to model the internal problem-solving mechanisms of ESs so as to make them available for explanation and examination is really the heart of explanation generation research and development. Therefore, much of the literature available on explanation addresses Function 1 type concerns.

2.5 Summary

This chapter has provided a common ground of understanding upon which to conduct a detailed examination of the work that has and is being done in the field of explanation generation. After providing a specific definition for the word *explanation* we looked at four reasons why EFs are important. These ranged from the academic desire to make machines claiming to be *experts* display the characteristics associated with human experts, to the establishment of an EF as the mechanism for instilling confidence and credibility in its expertise, providing a debugging and testing tool, and providing the means for instructing/teaching less knowledgeable users about the problem domain. We then identified and discussed the three functions that make up a functional framework for generating explanations. These include the concerns related to modeling the knowledge and reasoning process of the ES, determining how much of an explanation is needed by a given user, and deciding how to present the explanation to the user in a form that will be easily understood.

Chapter 3

MYCIN: The Beginning

3.1 Overview

The logical place to begin our examination of explanation generation research is at the beginning, with the ES, MYCIN [2]. MYCIN is a consultation system designed to diagnose and recommend treatment for infectious blood diseases. It is a rule-based production system that uses certainty factors to handle inexact, ambiguous information and backward-chaining (often times referred to as a hypothesis driven inference mechanism) as its control strategy.

We will examine this *birthplace of EFs*, these early beginnings of explanation generation, in four basic parts. First, a historical synopsis of MYCIN's development will be presented. This will be followed by a high level discussion of the first basic assumptions, general characteristics and goals, and major functional areas identified by MYCIN's developers as necessary for any EF. We will then examine the specifics of this *1st EF* and follow this examination with a discussion of the accomplishments and limitations that resulted from this pioneering work in explanation generation.

Buchanan and Shortliffe's book entitled Rule-Based Expert Systems [2] is a complete and comprehensive description of the entire MYCIN project and is the primary source of information for this chapter. However, the information I have extracted is but a small piece of this extremely significant work in ES research and development.

I highly recommend this book to any interested reader.

3.2 Historical Synopsis

The initial discussions that led to the MYCIN project were conducted in 1972 at Stanford University. These discussions involved researchers from the University's School of Medicine and Computer Science Department. Discussion centered on developing ways to add intelligence to computer programs that processed medical data. The two key players that emerged from this collaborative effort were a computer scientist named Bruce Buchanan and a medical student named Edward Shortliffe. In fact, the discussion of the identified problem and the computer system that was developed as its solution (MYCIN) became the Ph. D. dissertation of Shortliffe [30].

By 1973 Shortliffe had published his first journal article about MYCIN. At this time the basic knowledge representation and control strategies of MYCIN were relatively well developed. However, the EF existed in only a rudimentary state. Shortliffe had developed a RULE command that could display a specified rule onto his terminal screen. The purpose of this command was for debugging the knowledge base where the rules were simply **echoed** back to the screen in their original LISP code format. After using this command for a short while the design group realized that an English translation of a rule would provide an enhanced explanation of what that rule was doing, in addition to providing a partial justification for the existence/use of the rule. About the same time Shortliffe's first article was published, Gorry published an article about the work being done at M. I. T. on a program for diagnosing acute renal failure. In discussing the limitations of this system, Gorry identified the lack of an explanation capability as the most serious deficiency of the program. This article inspired Buchanan and Shortliffe, causing them to hypothesize that explanation was not only invaluable to knowledge engineers (as Gorry had

noted) but was also the critical link needed to provide the confidence and credibility a computer system needs in order to be used and accepted by physicians, a problem that existing medical assistance programs had. In fact, as Buchanan and Shortliffe state in their book,

> "We were especially convinced that explanation capabilities were crucial for user acceptance and that this single failing in particular largely accounted for the rejection of systems based solely on statistical approaches ... we could not prove that explanations would make a difference unless we implemented a consultation system in a clinical environment where controlled studies could be undertaken." [2, page 602]

This marked the beginning of explanation generation research. So important was it believed to be that from 1973 through 1975 it was the primary research and development focus of the group at Stanford University that became known as the *MYCIN Gang*.

3.3 Concepts and Fundamentals

The initial assumptions, concepts, principles, etc. that are developed by pioneers in a brand new area of research are often times very important because they form the foundation on which all subsequent research is based. This is especially true in explanation generation research. In an effort to establish this foundation with respect to our discussion, some of these high level fundamental ideas are now discussed.

3.3.1 Basic Beliefs

Four basic beliefs of the *MYCIN Gang* formed the foundation of explanation generation in ESs. [28]

1. A consultative rule-based system did not have to be a psychological model that precisely imitated a human's reasoning process.

2. The important goal was for the system and the human expert to use the same (or similar) knowledge about the application domain and arrive at the same (or similar) answer/decision about a specific problem.

3. The process of the system trying rules and taking actions was equivalent to the human reasoning process.

4. If information contained in the rules was sufficient to show why an action had been taken (omitting any programming details) then an explanation could simply display the rule verbatim or in a free-text translation.

These beliefs formed the basis on which the EF in MYCIN was designed and implemented. Remember, however, that these beliefs were not unique to this development effort but were believed by this team of researchers to be universally applicable (even though the universe of explanation generation in ESs was not very big at the time).

3.3.2 General Characteristics And Goals

The general characteristics and goals identified by the *MYCIN Gang* [28] have already been indirectly described, in Chapter 2. However, they are restated here as a necessary building block in the construction of the initial foundation of EF research.

1. The basic purpose or goal of an EF is to give the user access to as much of the system's knowledge as possible.

2. An EF must be able to handle questions about all relevant aspects of the system's knowledge and actions.

3. An EF must provide complete and comprehensive answers to the user.

4. An EF must be easy to use (for novice and/or expert alike).

3.3.3 Major Functional Areas

MYCIN's developers identified two primary functions a consultative expert system EF should perform.[28] The first they called the *Reasoning Status Checker* (RSC) and the second they called the *General Question Answerer* (GQA). While these names are unique to the MYCIN project, the functions associated with them are, again, ones that were believed to apply to explanation generation in general.

The RSC is used during the actual execution of the consultation session and is concerned with answering questions about the status of the system's reasoning process. Examples of these types of questions are:

- **WHY** is the system requesting a particular piece of information?
- **HOW** is a goal or subgoal achieved?

In order to answer these questions, the RSC must keep track of what the system has done. It needs to know what goals and subgoals were being addressed and which rules were fired to achieve a specific goal. The RSC also needs to have a general knowledge of how the rule interpreter works. Finally, the RSC needs to have a general understanding of the individual rules. It needs to know what each rule means (or know where to look for this meaning) and it needs to know what each rule is used for — the goal it is applied against.

The GQA is used during and **after** the consultation session and is concerned with answering questions about the current state of the system's knowledge base. Examples of these types of questions are:

- **WHAT** did rule X tell you about object Y?
- **WHAT** is the value of parameter X?
- **HOW** did you use the attribute value of object X?

The types of questions that can be asked of the GQA cover a much wider range of possibilities than the two types available for the RSC. Therefore, the GQA must be able to recognize questions from this much larger range of possibilities. This often involves the topic of *natural language processing*, which is a complicated and complex research area in and of itself. However, the natural language processing problem aside, there are several types of information (or knowledge) the GQA must have in order to do its job. Obviously, it must have all of the knowledge the RSC has. In addition, it must know how the static and dynamic information in the knowledge base is structured and stored.

The above constitute the basic concepts and beliefs identified by the MYCIN Gang and make up the foundation of EF research.

3.4 MYCIN's EF

As we look at how the developers of MYCIN incorporated the principles and concepts identified above into an actual EF, we will do so in more detail than for any other EF we examine. Again, this has to do with the fact that MYCIN's EF was the first one ever developed and is responsible for setting the standards for much of the subsequent research in this area.

3.4.1 Specialists

The developers of MYCIN's EF tried to anticipate all of the types of questions a user might ask of the system. They then created *specialists* (separate pieces of code) to answer each of these anticipated questions — each specialist providing one type of explanation. These specialists were grouped into three sets: those that explained what the system was doing at a given time, those that explained the system's static knowledge, and those that explained the system's dynamic knowledge. It was in these different sets of specialists that the various types of knowledge required by the

RSC and GQA were encoded.

3.4.2 Knowledge Organization

MYCIN's knowledge was organized into four basic structures. The first was the *attribute-object-value triple*, a common data storage structure. Second, *lists* were used to categorize and store static information. *Knowledge tables* were the third organizational structure and were used to provide comprehensive records of certain parameters and the values these parameters could have under various situations. These tables helped reduce the complexity of many of the rules by eliminating the requirement for rules dealing with multi-valued parameters to have to conditionally check each possible value of the parameter. Finally, there were *four specialized functions* that were used for comparing and manipulating the system's parameters. Each of these functions had a specific purpose and contained the necessary knowledge to fulfill that purpose.

3.4.3 Rule Structure

To provide a structure for grouping and categorizing the system's static and dynamic knowledge, MYCIN's developers created ten *context types*. In addition, they identified approximately sixty-five primitives or characteristics of these different context types. Each of these primitives had a set of properties associated with it that described such things as the legal value range for the primitive, the list of all rules that used the primitive, a translation of the primitive into its English representation, etc. These primitives constituted the *language* that was used to encode all of MYCIN's rules. By having a small set of well defined primitives and by using an appropriate template, MYCIN was able to provide English-like explanations to the user.

3.4.4 History Tree

The history tree is the record of interactions with the user and the trace of the system's performance (as required by the RSC and GQA). Because of MYCIN's backward-chaining control strategy, the history tree reflects this goal-directed solution approach. Each node on the tree is a goal (or subgoal) and contains information about how the system tried to accomplish that goal (i. e. Asked the user, or, Tried some rule(s)). In addition, each rule has a record of its success or failure and a reason why it failed, should that be the case. Therefore, one can see the wealth of knowledge contained in this tree. It is here that the individual specialists look for much of their information. By starting at the current node (the current goal being addressed) and traversing up the tree to its root the **WHY** questions of the RSC are answered. By starting at the root of the tree and traversing down to the current goal the **HOW** questions of the RSC are answered.

3.5 Accomplishments And Limitations

3.5.1 Accomplishments

The most important accomplishment of the MYCIN project is the identification and establishment of the need for ESs to provide a means of explaining themselves. Buchanan and Shortliffe were convinced that explanation facilities were crucial for user acceptance of consultative expert systems and set out to prove that this **gut** feeling was correct. In 1980 they conducted a formal analysis of physicians' attitudes concerning this subject. The details of this study are described in [2, Chapter 34]. However, the pertinent information from this study, with respect to our discussion, is that **the ability of a computer program to provide explanations about its reasoning process was determined to be the single most important requirement for a medical advice-giving system.**

Another important result of the MYCIN work is the foundation for explanation generation established by this project. The basic characteristics and goals identified for EFs are still considered applicable today. The structuring of knowledge, use of templates for English-like explanation generation, and the building of a history tree are all initial ideas that are still in use.

3.5.2 Limitations

Almost as important as its accomplishments are MYCIN's limitations, for it is the identification of these limitations that provides the fuel for further investigation and research. Subsequent chapters of this thesis will examine the research that has been conducted and the systems that have been developed to overcome or improve upon these limitations.

Hasling, Clancy, and Rennels [36] identify the first major limitation. MYCIN is unable to explain the strategy it used to solve a particular problem. This is primarily due to the fact that the strategy information is implicitly contained in the rule orderings of the system and is unavailable for explanation. When the knowledge engineers encoded the expert knowledge (the rules) certain rules were identified to be tried before others. This ordering was based on knowledge that was obvious to the expert (often times resulting from the expert's experience with the subject matter) but was not explicitly encoded into the system and therefore unavailable for explanation. Yet being able to explain this strategy adds credibility to the human expert and would do the same for a computer program.

Swartout [35] identifies a second limitation, one that is closely related to the first. This limitation is MYCIN's inability to justify its actions. The system can describe what it did by using its execution trace/history tree, but it can't explain why its actions are reasonable in terms of domain principles. Again, the reason for this is that this information is implicitly contained within the structure of the system and

can't be used by the EF.

A third limitation is identified by Rubinoff [26]. He notes that MYCIN can't explain its concepts and terms, explanations that are needed when a user is confused about what the system is asking.

The fourth limitation, identified by Swartout [34], concerns itself with MYCIN's inefficient handling of change and revision to a previously provided answer. While MYCIN readily accepted the changed answer, it would then recompute the entire consultation process thereby re-executing many unaffected steps and providing explanations to the user that contained large amounts of redundant, irrelevant information.

The final limitation, again identified by Swartout [34], concerns itself with the small knowledge limit conveniently expressable in a single rule. Swartout credits Randall Davis for making this discovery in his work on meta-knowledge (some of Davis' work will be discussed in the next chapter). When actions to be accomplished are larger than can be expressed in a single rule, the combination of several rules is used. Often times, accurately constructing and relating these rules to achieve the intended action is difficult to do.

3.6 Summary

In analyzing MYCIN with respect to the Functional Framework described in Chapter 2 we find that two of the three functions were addressed to some degree. The history tree provided the Type 1 explanations of the first function. However, Type 2 and 3 explanations were not provided. The combination of rule structure and template provided the requirement for Function 3. The English-like explanations of this first EF were quite an accomplishment.

MYCIN's original purpose was to provide a solution to the problem of adding

intelligence to computer programs that processed medical data. Early in its development stage, a second purpose was identified. The lead architects of the system wanted to prove that their gut feeling regarding the importance of explanation for user acceptance was correct. MYCIN has successfully fulfilled both of these purposes. More importantly, it has recognized and led the way into an extremely important area of ES research and development — the generation of explanations — providing the capability for an ES to explain how and why it does what it does.

Chapter 4

The MYCIN Family: TEIRESIAS, GUIDON, and NEOMYCIN

4.1 Overview

As identified in Chapter 3, explanation generation research became a primary focus of attention shortly after the initial MYCIN project began. This was due in large part to the numerous limitations identified in MYCIN's EF. The purpose of this chapter is to describe the various research efforts that resulted directly from the initial MYCIN system. All of the systems we are going to discuss in this chapter are directly related to MYCIN. For the most part they use the knowledge representation schemes, inferencing mechanisms, etc. already established in MYCIN. Assuming MYCIN to be Version 1 of EFs, these systems constitute Version 2 — the first upgrade to the initial system. As will be identified, the motivation for these systems varies. However, each was developed to increase the power and application of the initial system by enhancing its explanation generation capabilities.

4.2 TEIRESIAS

4.2.1 Background

TEIRESIAS was developed by Randall Davis [2, Chapters 9 and 17] to provide an automatic acquisition mechanism for the addition of new knowledge to MYCIN's knowledge base. It was designed as a front-end or ancillary system to MYCIN and was concerned primarily with the user interface needed to provide this desired capability. However, the need to provide enhanced explanation for debugging and testing of added knowledge was also recognized and provided the motivation for the research we are going to examine.

4.2.2 Contributions

There were two primary contributions to EF research that resulted from the TEIRESIAS project. The first was the concept of an *information metric* to help provide varying degrees of explanation detail to the user. The second was the formal definition of four steps needed to design an EF.

Function 2 of the Functional Framework described in Chapter 2 is concerned with presenting explanations to the user based on their individual needs and abilities. The *information metric* [14, page 269] developed by Davis was the first attempt at providing a capability in this area. Using the Certainty Factors (CF) already associated with the knowledge in the knowledge base, Davis chose **-(log CF)** to represent his information metric, his measuring device on how much or how little explanation to provide to the user. This value was not based on any formal theory or justification of meaningfulness and didn't account for the user's knowledge about the application domain. It was simply a *dial* available to the user to adjust the level of detail provided in an explanation. The way this metric functioned was also quite simple. The distance from the current node in the execution trace/history

tree to the top of this tree was computed and normalized to 10. The user, if he so desired, input a number that was taken as a fraction of this normalized distance and explanation from the current node to that level in the tree was provided. Again, this whole idea wasn't founded in any scientific theory. However, its implementation was considered a success.

The second contribution identified above was the four design steps Davis formally identified as being necessary to design an EF. [14, page 264] These four steps are:

1. **Determine the program operation that is to be viewed as primitive.** In the case of TEIRESIAS (and MYCIN) this primitive operation was the invocation of an individual rule.

2. **Augment the program code to leave a trace or record of the system's behavior at the level of detail chosen in (1).** The history tree of goals and rules provided this capability.

3. **Select a global framework in which the trace or record can be understood.** Davis chose a goal tree and provided a set of commands for this framework. This choice was based on the backward-chaining control structure used in MYCIN. However, different frameworks can be identified for different system designs.

4. **Design a program that can explain the trace or record to the user.** Implement the set of commands identified in (3).

As EF research has grown and expanded and as additional requirements for EFs have been identified, these design steps have also expanded and grown. However, as will be seen in the following chapters, these four steps are still relevant today.

4.2.3 Limitations

In designing TEIRESIAS's EF based on the above steps, Davis identified several limitations to the system. First, the choice of a single primitive operation and a single framework restricted the class of events that could be explained. In addition, only surface level (unsophisticated) explanation could be given. For example, the system only interpreted the meaning of **WHY** one way and provided its explanation based on this interpretation. However, there are several different interpretations of this word. TEIRESIAS's EF couldn't handle these different WHYs. Finally, Davis noted that no capability existed to explain the control flow or solution strategy of the problem solving process.

4.2.4 Conclusion

The primary importance of the TEIRESIAS project was involved with the automatic knowledge acquisition capability it provided to MYCIN. As this was beyond the scope of this paper, it was not discussed. However, this does not reduce the significance of the EF accomplishments we have identified above. As has already been mentioned, the formal design goals identified are still relevant today. While nothing in the literature indicates how effective the information metric was (except to say that it was **successful**), it is definitely a part of the MYCIN system and is mentioned frequently by those developing other ideas in this particular area.

In analyzing TEIRESIAS from the point of view of our Functional Framework described in Chapter 2, we find that this system provided MYCIN with an increased Function 2 capability, the information metric.

4.3 GUIDON

4.3.1 Background

MYCIN's success as a problem solver of infectious blood disease diagnosis seemed to indicate that its knowledge base was rich enough to provide a source of material for teaching students about this application domain. William Clancy was one of the individuals that believed this to be true and was the primary architect of GUIDON [2, Chapters 26 and 29] [10], a system designed to be a tutorial version of MYCIN.

4.3.2 Problems Encountered

Initially, Clancy felt that a simple restructuring of MYCIN's rules was basically all that would be needed to provide a tutorial capability. However, he soon discovered this was not the case. MYCIN was developed with expert users in mind (practicing physicians) — tutoring involves explanations to naive users (medical students). Much of the knowledge Clancy needed to have available for explanation was not there. For example, the expert diagnostic approach and understanding of the rules that was used to build the system was not explicitly encoded. The rules could not be justified because the concepts of how the rules fit together were not explicitly identified. The problem solving strategy could not be explained because the structure of the search space and the strategy for traversing it were implicitly contained in the ordering of the rules. Because of these missing pieces of knowledge needed to provide a tutorial program, Clancy (and others) conducted a detailed re-examination of MYCIN's rule base and the foundations upon which the rules were constructed. This re-examination resulted in identifying several important concepts and ideas that have subsequently been applied beyond the tutorial environment in which they were discovered.

4.3.3 New Concepts And Enhancements

The first concept identified was that of *strategy*. Clancy defined this concept as a plan by which goals and hypotheses are ordered in a problem solving process. This concept is important for an ES to be able to explain, but knowledge about this strategy must be explicitly provided in order to do so. A second concept was that of *structural knowledge*. Clancy defined this concept as abstractions that are used to index into the domain knowledge, to provide a *handle* by which a strategy can be applied. A third concept was that of *support knowledge*. Clancy defined this concept as low-level information that identifies the causal relationships between different types of structural knowledge. These three concepts, which Clancy identified as the *strategy/structure/support framework*, were a major break through in EF research.

Having identified this framework, GUIDON's developers then set about figuring out ways to encode this knowledge into the MYCIN knowledge base. The method chosen was to use domain-independent *meta-rules* to explicitly specify the problem-solving strategies. While a detailed discussion of *meta-knowledge* and/or *meta-rules* is beyond the scope of this paper (see [13] for complete details), in a nut shell, *meta-knowledge* is **knowledge about knowledge** and *meta-rules* are **rules about rules**. From an abstraction level point of view, domain knowledge and domain rules are the lowest, most detailed level of knowledge abstraction in the system. Meta-knowledge is at a higher level of abstraction and is used to explain the knowledge at the lower level. This was precisely what Clancy needed.

4.3.4 Conclusion

In analyzing the work done in this system with respect to our Functional Framework of Chapter 2, we find that all of it pertains to knowledge representation, a Function 1 application. Therefore, GUIDON enhanced the Function 1 capabilities of MYCIN.

The most dramatic discovery resulting from the GUIDON system was that one can t simply add an interactive front-end onto any AI program (especially an ES) and expect it to be an adequate and effective instruction mechanism. In order to be effective tutors, EFs must do more than just read back their reasoning steps and be able to recognize questions. They must be able to abstract the reasoning steps and relate them to the domain models and problem solving strategies they use. Clancy's identification of the *strategy/structure/support framework* in response to this problem has become one of the most significant motivators of explanation generation research in the field.

4.4 NEOMYCIN

4.4.1 Background

The developers of NEOMYCIN; Hasling, Clancy, and Rennels [36], hypothesized that an *understander* (as they called it) must have an idea of the problem-solving process and domain knowledge in order to understand and solve the problem himself. With this in mind, these individuals set out to develop a knowledge base that facilitated recognizing and explaining diagnostic strategies. Interestingly, this sounds vaguely familiar to the previous section on GUIDON. One readily recognizes Clancy's name on both projects. Not surprising then that much of the work on these systems overlapped.

4.4.2 Problems Encountered

Basically the problems encountered were the same as those encountered in GUIDON. Knowledge of the diagnostic approach and understanding of the rules were not available for explanation. Neither were the concepts of how the rules fit together or the structure of the search space or the strategy for traversing it.

4.4.3 Enhancements/Achievements

The first important achievement resulting from this project was the formal recognition of the need for understanding and being able to explain problem-solving strategies. Second was the realization that separating control knowledge from domain knowledge was extremely important, as was explicitly representing this control knowledge in abstract rules. Finally, and most importantly, was the development of the *task* concept. A *task* is a representation hierarchy of meta-level goals and subgoals. Remembering that meta-rules are used to explicitly encode the control knowledge into abstract rules, these meta-rules are the methods used to achieve the goals and subgoals in a task. These meta-rules invoke other tasks which use their meta-rules to achieve their goals and, in turn, invoke other tasks, etc. until ultimately the *grain level interpreter* is invoked to pursue domain goals using domain rules. This whole concept of tasks is an interesting one and will be discussed in greater detail in Chapter 6.

4.4.4 Conclusion

In viewing NEOMYCIN from our Functional Framework point of view we find the same application as in the GUIDON system. Strategic explanations deal with Function 1, Type 3 explanations.

Providing strategic explanations was the primary focus and importance of the NEOMYCIN project. Interestingly, providing these explanations is not much different than providing explanations about the application domain. The only difference is that strategic explanation takes place at a higher level of abstraction than domain explanation. Another interesting point is that understanding domain level concepts is an important prerequisite to understanding and appreciating strategic explanations.

4.5 Summary

This chapter has discussed three areas of explanation generation research that were based on, and the immediate result of, the initial MYCIN program.

TEIRESIAS, designed as a front-end knowledge acquisition interface to MYCIN, was concerned with user interaction. As a result, its contribution to EF research dealt with providing varying degrees of explanations based on the users' desires and input.

GUIDON, designed to expand MYCIN into a tutorial system for medical students, discovered that the original knowledge representation schemes in MYCIN were not robust enough to support a teaching environment (an environment that deals with naive/novice users). In addition, much of the knowledge needed to provide tutorial type explanations was implicitly hidden in the knowledge base and therefore unavailable for explanation. These deficiencies in the original system motivated extensive analysis and research which led to the development of the *strategy/structure/support framework*.

NEOMYCIN was simply a second generation or second version MYCIN program developed to provide explanations of problem-solving strategies. The major contribution here was the separation of control knowledge from domain knowledge and the concept of similar explanation processes at two different levels of abstraction (the task level and the rule or domain level).

This chapter and the information presented in Chapter 3 constitute a review of the first work done in explanation generation research and development. Much of what has been discussed is still available in expert system EFs today, even though the ideas and concepts are ten to fifteen years old. However, more recent avenues of EF research present some new and different ideas and are the focus of the next several chapters.

Chapter 5

Explanation & Explicit Development Models

5.1 Overview

In Chapter 3 we noted that a M. I. T. article by Gorry provided the initial inspiration for the EF research conducted in the MYCIN family of projects. The importance of and need for explanation, therefore, existed in the minds of M. I. T. researchers as well as in the minds of those at Stanford. This chapter looks at some of the work done in explanation generation at M. I. T. Specifically, it looks at the work done by William Swartout over a ten to twelve year period (late 1970s to present) involving three related expert systems: *A Digitalis Therapy Advisor With Explanations* [34], *XPLAIN* [35], and *Explainable Expert Systems (EES)* [24,32]. By examining each of these in detail we will recognize the important enhancements and extensions to explanation generation this total research and development effort has contributed.

5.2 A Digitalis Therapy Advisor With Explanations

5.2.1 Background

The Digitalis Therapy Advisor With Explanations project (which we will refer to simply as Digitalis) was begun sometime in 1977 for the purpose of providing advice

to physicians regarding digitalis therapy. Therefore, this system was a medical consultation system similar to MYCIN. Digitalis was based on a previously developed digitalis advisory system that had been done at M. I. T. by Pauker, Silverman, and Gorry. Swartout's work was concerned with providing an EF for this system (as its name indicates).

5.2.2 Problems Addressed

There were three main problems or limitations in existing EFs that motivated Swartout's work on Digitalis. Two of these were identified from the MYCIN system: one being revision and the other being the limited information content capability of an individual rule. The third problem comes from the Software Engineering area. Swartout recognized that an explanation system could be used to provide self-documenting programs and, thus, help alleviate the well known documentation problems that exist in software development and maintenance.

5.2.3 A Procedural Approach

Based on Davis' work with meta-level knowledge and rule based systems [12], in which he recognized that physicians tend to think of sequences of operations in procedural terms, Swartout chose *procedures* rather than *rules* as his primitive knowledge representation structure. Using the top-down refinement approach of standard *structured programming*, Swartout developed Digitalis as a hierarchy of procedures. This hierarchy was broken into levels of abstraction, with higher level procedures representing higher level goals and lower level procedures representing lower level subgoals. As will be seen shortly, developing this hierarchy of procedures provided some distinct advantages. Additionally, Swartout included the now familiar execution trace/history tree in the design of Digitalis.

5.2.4 Contributions/Enhancements

The first major contribution of Swartout's approach was that it more closely modeled the structure of a human expert's solution process. In fact, Digitalis's success in providing enhanced explanations was fundamentally based on this assumption. If the model of the program's solution structure was the same as (or close to) that of the human expert then it was logical to assume that the explanations generated from such a model would be more understandable by the user.

The second major contribution was that the levels of abstraction in the hierarchy of procedures provide a natural (implicit) way of providing varying degrees of explanation detail to the user. Swartout provided the capability to explain every procedure in the system. In addition, higher level procedures could summarize all of the calls it made to lower level procedures. Thus, the first request for explanation a user made yielded the higher level summary. For each subsequent request by the user, the next lower level in the hierarchy was explained. This process continued until the user had received the degree of explanation detail desired.

The third enhancement of Digitalis addressed the revision problem. Remember that when MYCIN was ask to recompute its answer based on different input information, it had to recompute the entire consultation session. Swartout overcame this inefficiency by developing a *smarter* interpreter that only recomputed the steps effected by the changed input. Thus, performance efficiency increased and the amount of redundant information provided in the explanations decreased.

The final enhancement we will discuss involves Digitalis's ability to explain how program variables were set and used. Because Digitalis's knowledge base was completely cross-referenced, Swartout developed routines that took advantage of this cross-referencing and provided these additional explanations.

5.2.5 Limitations

The first limitation was based on the assumption identified earlier — that the program's structure closely matched the human expert's structure for solving the problem. If this wasn't true, Digitalis couldn't provide very useful explanations. Therefore, a great responsibility was placed on the system designer to insure the above assumption was met.

The second limitation results from using the hierarchy of procedures. Digitalis could only provide a limited number of explanations. This number was directly related to the number of levels of abstraction in the hierarchy and was fixed once the program was written. If the explanation at one level was too general and at another level was too detailed, there was no way to produce intermediate levels of explanation if that abstraction level did not already exist.

Another limitation of this system was its inability to tailor its explanations to its user. Digitalis did not attempt to obtain a specific model of each individual user. Explanations were provided in one format whether the user was a layman or a doctor, a novice or an expert. In addition, because an English parser was never developed, requests for explanations had to be entered in the form of LISP function calls.

Finally, Digitalis couldn't adequately justify its actions. It didn't have the knowledge it needed explicitly encoded and available for explanation.

5.2.6 Conclusion

In analyzing this system from the standpoint of our Functional Framework we find that Digitalis provides the same Function 1, Type 1 explanations as MYCIN did and used the same mechanism for doing so — the execution trace/history tree. Its Type 2 explanation capabilities were found in its use of the cross-reference index and

in its ability to explain each procedure in its code. Digitalis could provide limited justifications for its reasoning but still fell considerably short of the full intent of Type 2 explanations. As with MYCIN, Digitalis didn't provide any Function 1, Type 3 explanations at all. Digitalis's Function 2 capabilities were implicitly contained in the levels of abstraction of its hierarchical structure. Digitalis also provided its explanations in both algorithmic form and in English, a Function 3 capability. However, its requirement for input to be in the form of LISP functions was a definite drawback.

This first EF by Swartout made several improvements over MYCIN's original EF and was ultimately successful in providing solutions to the problems that motivated its development. However, its significant importance is more as a stepping stone for further research (similar to MYCIN) than as a functional EF.

5.3 XPLAIN

5.3.1 Background

The XPLAIN system [35] was developed several years after Digitalis and was a direct descendent of this system. Its application domain (digitalis therapy) and functional purpose (providing physicians with advice for administering digitalis therapy) were the same and it used many of the original Digitalis structures and concepts.

5.3.2 Problems Addressed

Swartout was convinced that the next important step in EF development was for a system to not only explain **what it did** but to explain **why it did what it did** -- to justify its reasoning process. He felt this justification was important because it could reveal whether or not a system was based on sound principles or was being pushed beyond the bounds of its expertise. Realizing that the knowledge needed to provide this justification was the knowledge required by the programmer to create

the program, but not required for successful performance of the program once it was written (our **deep** knowledge again), Swartout searched for ways of capturing and explicitly encoding this knowledge. XPLAIN was developed as a solution to this problem.

5.3.3 An Explicit Development Approach

Again, the motivation behind XPLAIN's development was Swartout's desire to discover some way of capturing the programmer's knowledge and decisions that were used to create the program, to capture this **deep** knowledge. What Swartout came up with is really quite interesting. He decided to develop an *Automatic Program Writer* that would use several different sources of knowledge to create a performance program of the system, keeping a record of its decisions in the process. There were five basic parts to the system he designed to do this:

1. *Domain Model*: The domain model represents the facts of the domain. It is a model of the causal relationships that are important in digitalis therapy. These facts correspond to the sort of facts a medical student would learn during the first two years of medical school. These facts tell **what happens** in the domain but do not tell **what behavior** should be used to achieve the goal of administering digitalis.

2. *Domain Principles*: The domain principles tell how something is to be done and can be thought of as *abstract procedural schemas*. These schemas are filled with facts from the domain model to produce specific instantiations of procedures. Domain principles are therefore comprised of variables and constants. There are four parts that make up a domain principle:

   (a) *Goal*: The goal of a principle indicates precisely what this principle can accomplish. Domain principles are arranged in a hierarchy based on the

specificity of these goals.

(b) *Domain Rationale*: The rationale of a domain principle consists of a pattern that is matched against the domain model and is used to provide additional information for achieving a principle's goal.

(c) *Prototype Method*: The prototype method is an abstract method for accomplishing a domain principle's goal. It is instantiated each time the domain rationale matches the domain model. When this happens a new structure is created with the variables in the prototype method being replaced by their matching counter-parts from the domain model.

(d) *Constraints*: An optional set of constraints may be attached to a domain principle and are used to identify specific events or criteria that must be met before the principle can be used.

3. *English Generator*: The English generator, which provides the English-like explanations presented to the user, consists of two parts.

(a) *Phrase Generator*: The phrase generator constructs English-like phrases directly from the knowledge base and is the primary work horse of this part of the system.

(b) *Answer Generators*: Answer generators are responsible for selecting the different parts of the knowledge representation structures that are to be translated into English by the phrase generator. These routines are responsible for determining the level of detail of an explanation. This is done using *viewpoints*, which are indicators attached to the steps in the prototype methods and identify who each particular step should be explained to (i. e. novice, knowledge engineer, domain expert, everyone, etc.).

NOTE: The phrase generator and the knowledge representation language used in this system were not developed by Swartout. They were contained in a system called XLMS (eXperimental Linguistic Memory System) which had been previously developed at M. I. T.

4. *Automatic Program Writer*: This is the program that uses the domain model and the domain principles to create the performance program for a particular digitalis advisory session. The user of the system actually interfaces with the performance program during execution.

5. *Refinement Structure*: The refinement structure is the most important part of Swartout's system because it contains the **deep** knowledge Swartout set out to capture. The refinement structure is the trace left behind by the automatic program writer showing how the writer developed the current digitalis advisor performance program. This structure is actually a tree of goals, each being a refinement of the one above it. At the lowest level of this tree (its leaves) are the primitive operations that the performance program actually executes.

The explanations XPLAIN produces are generated using the refinement structure, an execution trace created by the performance program, the domain model, the domain principles, and the English generator. While details of how this is done exceeds the scope of this paper (the interested reader is referred to [35]), the important points to note are the development of additional types of knowledge structures (than seen in previous systems) and the creation of an interesting way of capturing the different types of knowledge that go in these structures.

5.3.4 Contributions

The most important contribution of Swartout's work in XPLAIN is the expansion of EF capabilities to another level of detail. The previous systems concerned

themselves with explaining the level of detail of how these systems performed their reasoning in arriving at a decision or answer. XPLAIN's ability to justify the use of this reasoning process moves into another (higher) level. As will be discussed more explicitly in the next chapter, providing this second level of explanation is an important step in expanding the range of problems ESs can be used to solve. Many people believe that if ESs are ever going to be used to solve really **significant** problems they must be able to produce high level explanations of their actions. Swartout's work in XPLAIN has taken a major step in this direction.

Another contribution of XPLAIN is the establishment of a domain-independent set of tools for generating explanations. The only changes that would have to be made to this system in order to **re-tool** it for a different application domain are the *domain model* and the *domain principles*. Everything else (the automatic program writer, the English generator, etc.) could remain unchanged and the resulting explanations should be as good as those in the original system. While there was nothing in the literature to indicate this was done, it was definitely mentioned as a possibility.

One final contribution of the work on XPLAIN is the increased thought and consideration that must be given to the performance program (on the part of its designers/developers) in order to make this system work. While this may initially be seen as a complicating factor and therefore a limitation versus an advantage, Swartout argues that it is a definite advantage because this increased thought process produces a much better understanding of the problem to be solved. This, in turn, helps provide more efficient solutions, eliminates many of the more subtle errors and, in short, produces a better solution to the problem.

5.3.5 Limitations

XPLAIN also has its limitations. The first of these is that the automatic program writer was limited to a goal/subgoal refinement strategy. If a domain principle could not be found to refine a certain goal, the system could not progress any further. It did not have the ability to re-evaluate the troublesome goal and attempt to achieve it using some other strategy.

Another limitation was that XPLAIN could only provide a small set of different types of explanations. This was because the different types of explanation were produced by fixed procedures that were hardcoded into the system. No mechanism for adding more of these procedures was available.

5.3.6 Conclusion

The desire to provide a justification mechanism in an EF motivated the research and development of this new expansion into another level of explanation generation. XPLAIN was successful in providing a solution to this justification problem. In analyzing it from our Functional Framework perspective of Chapter 2 we notice several improvements.

Function 1, Type 1 explanations are still provided using the execution trace/history tree structure developed in the original MYCIN project. However, the refinement structure produced by the automatic program writer provides a mechanism for providing Type 2 explanations. It is important to note that while this paper has given credit to previous systems for providing some Type 2 explanation capabilities, with respect to the full definition Chandrasekaran [8,9] gives for this type of explanation, XPLAIN is the first system to provide such a capability. As for Function 1, Type 3 explanations, XPLAIN does not have the capability to provide them.

In looking at its Function 2 capabilities we find that XPLAIN has expanded on

this area using its answer generators and the viewpoints attached to the different steps in the prototype methods of the domain principles. If the user is an expert, all viewpoints identified as applying to an expert are recognized and their corresponding steps are explained. Likewise for a novice or other category of user.

XPLAIN also enhances the Function 3 capabilities seen so far. The English generator it uses is a much more elaborate system than the template mechanism used in MYCIN or the reliance on an English-like implementation language (as in the original Digitalis system).

5.4 The EES (Explainable Expert Systems) Approach

5.4.1 Background

EES [24,32] is a direct extension of the work done on XPLAIN. The motivation for this continuation effort is best described by the following, taken from an article by Neches, Swartout, and Moore.

> "The primary goal of the EES project is to provide a framework that will facilitate construction of expert systems according to a paradigm of separate models and recorded developments. This requires pursuing the lessons of the XPLAIN system by extending the forms of knowledge known, extending the kinds of development recorded, and extending the benefits beyond explanation to development and maintenance." [24, page 1340]

5.4.2 Problems Addressed

EES addresses the limitations identified in the proceeding section on XPLAIN. However, as can be seen from the quote just presented, this was not the driving factor behind EES. The real problem EES addresses is the logical extension of expanding a concept or technique initially developed in a constrained environment into a more general, robust one. Often times this expansion identifies completely different application areas where the concept or technique can be used, as well as expanding

the init ai application area. This is true with EES. The work done on this system not only benefits EF research and development but also provides assistance to the Software Engineering areas of software development and maintenance.

5.4.3 Enhancements

We will look at the enhancements EES makes to XPLAIN based on the three areas identified in the quote above: extending the forms of knowledge, extending the kinds of development recorded, and extending the benefits beyond explanation.

In addition to XPLAIN's domain model and domain principles, five other forms of knowledge are used in EES.

1. *Tradeoffs*: Tradeoffs are associated with domain principles and are used to indicate the beneficial and harmful effects of selecting a particular strategy (method) for achieving a goal.

2. *Preferences*: Preferences are associated with the goals of domain principles and make up a set of priorities for strategy selection based on the identified tradeoffs.

3. *Terminology*: Terminology is a new name for what was identified in XPLAIN as a domain principle's prototype method. In EES these prototype methods were broken out as a separate knowledge type so they could be shared across different domain principles.

4. *Integration Knowledge*: Integration knowledge is used to resolve potential conflicts among different knowledge sources.

5. *Optimization Knowledge*: Optimization knowledge represents ways of efficiently controlling execution of the performance system.

With respect to extending the kinds of development recorded, EES does this by redesigning the automatic program writer into two parts. The first part uses all of the different knowledge types (except the optimization knowledge) to produce the initial performance program just as XPLAIN did. The refinement structure that results, however, has been greatly improved because more and different types of knowledge were used to generate it and because the automatic program writer had the capability to reformulate a goal automatically if no domain principle could be found to refine it. The second part of the automatic program writer (the new part with respect to XPLAIN) modifies the initial refinement structure by running it through an efficiency optimizer (which uses the optimization knowledge). Therefore, the extended kinds of development recorded in EES's refinement structure are the result of more knowledge types, automatic goal reformulation, and optimization.

The extension of benefits beyond explanation was a natural result of the other two enhancements just identified. EES has access to many different kinds of knowledge and the ability to explain each of these types. Therefore, if one were to look at EES through the eyes of a software engineer, one would see the beginnings of a self-documenting system. As has been well established, documentation is a very serious problem with any large software program, especially after it has been changed and revised (maintained) several times. A system with an EES explanation capability would have its own self-contained documentation system. The new program maintainer could sit down at a terminal and ask questions to determine how the code works, why certain things were done certain ways, why or how specific pieces of data relate to other pieces of data, etc.

5.4.4 Limitations

The biggest limitation or area for future work in EES, as identified by its architects, is in the area of the user interface. Because EES provides so much EF capability,

the easier it is for someone to use, the more use it will get. Currently, it isn't easy to use.

5.4.5 Conclusion

EES has built on the lessons learned from XPLAIN. We have discussed how it did this with respect to the quoted statement at the beginning of this section. With respect to our old friend the Functional Framework, EES does not provide any startling new capabilities, but that wasn't it's purpose. It's basic purpose was to extend a technology developed in a very specific application domain into a general framework for expert system construction. How well EES has done this has yet to be determined.

5.5 Summary

This chapter has examined the explanation generation work done at M. I. T. over the past ten to twelve years. As with MYCIN and the family of related systems resulting from it, the work discussed in this chapter can also be viewed as a family of related systems.

The Digitalis Therapy Advisor With Explanations was the first system developed in this family. Representing knowledge as procedures instead of rules, placing these procedures into a hierarchy of abstraction levels, and providing the capability to explain every procedure in the system were the major contributions of this system.

XPLAIN was the second system developed in this family and was primarily concerned with providing justification type explanations. The automatic program writer and resulting refinement structure used to provide this type of explanation was a major contribution to the field of explanation generation research. In addition, in providing these justifications, a higher level of explanation was achieved, thus opening the door for a wider, more significant range of problems to be addressed by

ES builders.

EES, the last system developed in this family, provided the logical expansion and improvement of the XPLAIN system into a more generic, domain-independent ES development tool.

The work done with respect to this family of ESs is an ongoing effort. EES is a very current system providing state-of-the-art explanation capabilities. Interestingly, it is so current that little has been written about it in the literature. However, the significance of its underlying system (XPLAIN) is well documented.

Chapter 6

Generic Tasks & Explanation

6.1 Overview

The work we are going to examine in this chapter is a continuing effort being conducted at the AI Research Laboratory at Ohio State University. It started in the late 1970s under the direction and leadership of B. Chandrasekaran. As a result of their early work in medical diagnostic knowledge-based systems (work stimulated by the MYCIN project) the beginnings of a general framework for developing ESs began to be identified by Chandrasekaran's group of researchers. As their work in this area continued, it became evident that the development of this framework could unify three major areas in ES construction: *the problem-solving process*, *deep cognitive models*, and *explanation*. Further research and development of this framework continues to be a primary focus of Chandrasekaran (and those associated with him) and is now known as the *generic task* approach to knowledge acquisition and ES construction.

This chapter will examine this concept a little differently than has been done in previous chapters. No one particular ES (or family of ESs) will be discussed. Rather, we will first present an overview or summary of the entire generic task concept and will then identify the benefits it provides with respect to explanation generation. It is important to note that different experimental ESs have been developed as part of

this research and development effort. In addition, the documentation of this work is contained in numerous articles ([4,6,7,9,21,8,23,29] to identify a few), with the information presented in this chapter being taken from [6,9,8]. The other references are included for those interested in more detailed discussions of particular aspects of this overall concept.

6.2 Problem(s) Addressed

The primary problem that motivated this research effort is one that has been hinted at previously in Clancy's work on NEOMYCIN and Swartout's work on XPLAIN and EES but not mentioned or identified explicitly. Chandrasekaran et al. felt that ESs needed to explain themselves at a higher level (the architectural level) in a manner appropriate to the problem solving task being performed. However, they recognized that the current approaches to knowledge-based system construction were being conducted at too low of a level (i. e. the rule level) and that this made it difficult to provide high-level explanations. Due to the implementation languages available, system designers had to transform problems into these low level implementation languages and then turn around and provide explanations that required translating these low level representations back to the problem level. Therefore, the desire to build knowledge-based systems that contained basic explanation constructs at the appropriate levels of abstraction was a big motivator.

6.3 The Generic Task Framework

The basic assumption underlying this framework (and identified during the early medical diagnostic research) was that there are different types of problem-solving processes and that there are different types of knowledge and control structures used within each of these different problem-solving types. For example, *diagnostic*

reasoning is one type of problem-solving application and has its own specific knowledge types and control structures. *Theory formation*, *design*, and *perception* are other problem-solving application areas whose solutions have been attempted by ESs. Each of these also has its own specific knowledge types and control strategies. In addition, it is also assumed that, for the most part, the knowledge types and control regimes are different for each type of problem-solving area.

Based on this assumption (or these assumptions) the basic concept that was formulated was to represent the appropriate knowledge and control information for each different problem-solving task using a vocabulary appropriate for that particular task. This was to be done by first identifying the different tasks, second by studying and analyzing the knowledge and control structures used by each, and then by constructing a design language for each task that provided the knowledge structures and control strategies specific to that task. By doing this, the ability to program at a higher, more appropriate level of abstraction could be conducted (the same basic reasoning that motivated the development of the high level programming languages used in structured programming).

Use of this concept has identified six generic tasks, each with its own design language. A brief description of each is now presented.

1. *Hierarchical Classification*: This task is very applicable in diagnostic type applications as evidenced by our discussions of other systems that used a goal/subgoal hierarchy. It is based on the construction of a classification hierarchy and uses an **establish and refine** strategy (similar to the one described in EES) to control its processing. Each node in the hierarchy, starting from the top, tries to match its establishment conditions against data in the knowledge base. If successful, the process repeats using the successors of this node. If unsuccessful, the node and all of its successors are rejected.

2. *Design By Plan Selection And Refinement*: This task is used to design some object according to a set of specifications. Specialists (which are the software representations of components of the object to be designed) are arranged in a hierarchy that mirrors the structure of the actual object, with each specialist containing plans the system can use to determine when and if to use it. Using a top down control strategy, the plan of a specialist is chosen based on some specification. The design parts of the specialist are then instantiated and another specialist is called to fill in some other design part(s). This process is done recursively until the design is complete. If a particular design part fails to get instantiated, control moves back up the hierarchy to allow higher level specialists to change their plans and then the downward control is resumed. This provides a second chance mechanism for instantiating the part that initially failed.

3. *State Abstraction*: This task is used to examine the effects of applying changes to some type of system and is quite similar to the design task just explained. A hierarchy of system/subsystem components is constructed. However, control is from the bottom up in this task, not from the top down.

4. *Knowledge-Directed Information Passing*: This task is structured as a hierarchy of frames and is used to obtain attribute values of related data. This is accomplished by using *actual values for the attribute* (if they are known), using *frame inheritance* to infer their values from parents or children, or by executing *demons* that query other frames in the hierarchy for the desired information.

5. *Hypothesis Matching*: This task is used to match a concept, hypothesis, specialist, or whatever name a particular entity may have, against relevant data

in the knowledge base and to determine the degree of fit for the attempted match. The degree of fit is an abstraction of the data computed for use in the matching process to deal with the uncertainty so often associated with the data and is represented in symbolic form (i. e. high, medium, low, good, bad, etc.). This task is frequently used in conjunction with other tasks in a problem-solving process.

6. *Abductive Assembly*: This task is a different way of performing the diagnostic process associated earlier with the classification hierarchy. The basic idea is to build the **best** hypothesis possible that explains the data using causal knowledge that relates the set of possible hypotheses with the relative significance of the data items.

These six generic tasks do not constitute a closed or complete set. The search for additional tasks continues. However, as was stated earlier, design languages for each of these six tasks have been developed and several experimental systems have been built using them.

6.4 Contributions

There are several contributions that have resulted from this work. First, a formal methodology or framework has been defined for building ESs and producing better explanations. This framework provides all three of the Function 1 type explanations defined in our Functional Framework (this is not surprising, however, as the definitions of these types came out of this research effort). Specifically, Type 1 explanations are provided using the familiar execution trace/history tree. Type 2 explanations are provided in the different knowledge types defined for the specific task being performed (similar to the mechanism described in EES). Type 3 explanations are provided by the same execution trace used to provide Type 1 explanations.

This is because the control strategy is explicitly represented in the implementation language specifically designed for the task being performed.

A second contribution, and in many ways the most important, is the recognition that explanation can be derived **naturally** from deep models and an understanding of the problem-solving task. While this notion has been hinted at in discussions of previous work (specifically NEOMYCIN and EES), it has been formally recognized and established in this work on generic tasks.

Two other contributions have resulted from this work. First, by providing a framework for producing high level explanations, the door has been opened for the research and development of ESs capable of solving more significant, meaningful problems (further extension of this same contribution identified in Chapter 5). Second, this framework provides a set of generic tools for ES development.

6.5 Limitations

Chandrasekaran notes that while generic tasks are a vast improvement over the low level implementation languages, explanations can still be provided at a higher level of abstraction than the task level. This higher level is the level of the actual problem-solving task. For example, in medical diagnosis this level would be the diagnostic level and would provide explanations on how system actions and decisions met diagnostic goals. Therefore, generic tasks are not the final answer.

6.6 Summary

The emphasis of this chapter has been on explaining the conceptual framework developed at Ohio State for building ESs and producing high level explanations. The major contribution of this work is the recognition that if enough study and analysis of a problem-solving task can be conducted and the knowledge types and

control strategies used by the task identified and represented, then the language built to specifically provide these capabilities will inherently provide the capability to produce high level explanations.

Chapter 7

A Potpourri Of Additional EF Topics

7.1 Overview

The purpose of this chapter is to present additional work being done in explanation generation. The systems and work described here are independent of each other and of any of the previously discussed systems. They are grouped together in one chapter because each is its own individual system designed to provide a very specific capability versus the much broader range of related systems and work presented in the previous chapters.

The first system we will examine is concerned with providing enhanced Function 2 capabilities (from our Functional Framework). The second system presents some interesting work that has been done in analyzing and identifying the information content of typical rules in a rule-based system. The third and final system is concerned with identifying ways to provide enhanced EFs to existing expert systems in use in the field (non-research or non-academic type ESs). Interestingly, these last two systems are related in that they are both concerned with providing enhanced explanation capabilities to existing systems, though from much different perspectives.

7.2 BLAH

7.2.1 Background

The BLAH system [37] was developed by J.L. Weiner at the University of New Hampshire in the early 1980s. This work was conducted around the same time as TEIRESIAS and The Digitalis Therapy Advisor and was primarily focused on developing ways to structure explanations so that they were less complex and easier to understand. Much of the work done in BLAH was based on a series of psycholinguistic studies that analyzed the different ways humans explain decisions, choices, plans, etc. to one another. BLAH was then designed from the frame of reference of replacing one of the two individuals involved in this explanation process.

7.2.2 Features

The BLAH system was basically a production rule system made up of sets of rules and assertions. Attached to each assertion was a justification structure consisting of three parts. The first was the *justification type* which corresponded informally to one of several types of justifications that people use in explanations. The second part was a *list of all assertions* that justified **belief** in the assertion in question. The third part, which was optional, was a *list of all assertions* that justified **disbelief** in the assertion in question. This justification mechanism played an important part in the BLAH system.

Another interesting feature of the system was the way it segmented or partitioned its knowledge base. The first and most important of these (from an explanation standpoint) was the defining of two different views: the *system view* and the *user view*. Each view only contained information that its owner (the system or the user) knew about. The knowledge base was also partitioned according to groups of assertions that were related to one another in some way. Finally, the knowledge

base was broken into *worlds* which identified different sets of premises.

7.2.3 Contributions/Enhancements

Remembering that the primary focus of attention was to uncomplicate complex explanations so as to make them easy to understand, several interesting ideas were implemented in BLAH to provide this capability. We will examine four of these ideas from a conceptual level.

1. *Presupposition*: The idea behind presupposition is to limit what is explained to the user based on what he already knows. This was done in BLAH by deleting from the explanation tree any assertions that were contained in both views (system's and user's) of the knowledge base.

2. *Structure*: This concept was concerned with structuring an explanation so that it was presented in the most understandable form. This was accomplished by either rearranging the explanation tree/reasoning tree produced by the reasoning component or by breaking this tree up into components and presenting the explanation to the user in increments.

3. *Alternatives*: When alternative explanations existed in the reasoning tree, rather than explain each alternative separately, the system would alternate between giving part of the explanation for one alternative and then jumping to the other alternative and explaining the related part of that alternative. The purpose of this was to group related parts of alternatives together in order to show the relationship between them more clearly.

4. *Counterfactuals*: A counterfactual is an assumption the system makes when the user wants the system to show a particular item X and that item is not provable given the current state of the knowledge base at the time of the user's

request. When this happens, the system assumes that the user would like to know "*why not* item X" as well as to understand under what circumstances item X could be proved.

Providing these concepts in his system, Weiner created a system that was a leader in providing Function 2 explanation capabilities (referring back to our Functional Framework again).

7.2.4 Limitations

Most of the limitations identified here were ones that existed in other ES explanation facilities of that time. Things like the lack of a natural language generator, unfriendly user interface, and the lack of a justification capability using **deep** knowledge were common limitations in the early days of explanation generation.

One limitation unique to BLAH was Weiner's recognition of a limited user's view and the increased capabilities the system could provide if this view were increased.

7.2.5 Conclusion

BLAH was developed in the earlier stages of EF research and development and was instrumental in identifying several interesting ways to provide more **user friendly** explanations. Considerable study of human explanation patterns formed the basis for much of the work done on this system. The enhancements it made to Function 2 type explanations are quite significant.

7.3 CLEAR

7.3.1 Background

The CLEAR system was developed by Robert Rubinoff [86] in the early to mid 1980s at the University of Pennsylvania. It was designed as a front end system capable

of interfacing to an independently developed ES and was primarily concerned with computing explanations automatically from the rules in the ES. Interestingly, the main emphasis of CLEAR's explanations were directed a little differently than we have seen so far. While these explanations were directed at the user, their purpose was to explain the system's concepts and terms when confusion arose on the part of this user about what the system was asking for or was presenting at the time. Therefore, CLEAR's explanations were much more concerned with the question of "**WHAT** is meant by X?" than with the familiar "**WHY** did X?" and/or "**HOW** did X?" type questions.

7.3.2 Methodology

As already stated, Rubinoff was interested in computing explanations automatically from the rules of the knowledge base. He believed that rules could explain different concepts based on the type of information encoded in them. This led to an intensive study of rules and their content and resulted in the definition of several types of rules (particularly inference rules) and several types of concepts that could be found in different parts of these rules. The key to the CLEAR system's explanation capability was based on identifying what type of inference rule was being used and what concepts were encoded in its premise and conclusion. Before we look at the types of rules and concepts Rubinoff identified, remember that the emphasis was on **explaining the system's terms and concepts**, not its control strategy or decision making process or anything else. Therefore, when we say *explanation* in the following definitions we are referring to this more restricted meaning of the word.

1. *Types of Inference Rules*

   (a) *cause-effect*: These rules are common in medical diagnosis and are used to deduce a cause from an effect or vice versa. They identify the causal

relationships in the domain knowledge. These rules can be very useful for explanation provided the user understands the causal process behind the connection.

(b) *paraphrase*: These are rules where the conclusion means the same thing as the premise. For example, if X is a man and X is not married then X is a bachelor. These rules are very useful in explanation because they are based on the meanings of terms.

(c) *problem-action*: These rules infer the need for some action based on having detected some problem. They do not implement the action, simply assert the need for it. These rules can also be quite useful for explanation provided the user can see how the action is a response to a problem.

(d) *matter-of-fact*: These rules indicate a conclusion that is empirically true. There is no known underlying principle that explains why it is true, it just is. Generally these rules are not very useful in explanation because they only provide miscellaneous facts.

(e) *screening clauses*: Screening clauses are not really rules at all, but instead are clauses in the premise of a rule used to determine if the system should try and use the rule. These clauses, while very important to the execution of the system, are useless from an explanation standpoint and may be safely omitted from any explanation that may be using the rule because it contains other important information.

(f) *control*: These rules are nothing but screening clauses used in the control strategy of the system. As with screening clauses, they are useless in the explanation of a system's terms and concepts.

2. *Types Of Concepts That Appear In A Rule Premise*

(a) *trigger*: The rule only succeeds if some attribute has a particular value.

(b) *source*: The rule depends on the value of some attribute in the concept. Therefore the concept is the source of information used by the rule.

(c) *premise property*: This is an attribute which must have a specified value for the specified object(s) in the clause in order for the rule to succeed.

(d) *deduced value*: A value which the system deduces in the course of proving a premise clause.

(e) *chain value*: A value deduced while proving one premise that is to be used as the source for another premise.

3. *Types Of Concepts That Appear In A Rule Conclusion*

(a) *result*: The conclusion asserts that the concept is the value of some attribute.

(b) *inferred property*: An attribute whose value is asserted.

(c) *subject*: The rule deduces something about this concept.

4. *Types Of Concepts That Appear In Both The Premise And Conclusion Of A Rule*

(a) *passed-forward value*: Something that appears as a value in both the premise and conclusion, something being passed forward from conclusion to premise.

These definitions form the basis for CLEAR's explanations. Surprisingly, the process that produces the explanations is rather simple. After receiving a request from the user for a clarification of some term or concept all rules are analyzed and ranked based on their relevance to the term or concept in question. The ranking of

the rules is based on the location of the term or concept in the rule. This ranking process is now listed in order of importance from highest ranking to lowest:

- triggers and premise properties
- sources and inferred properties
- results and deduced values
- subjects

The highest ranking rules are then used to generate the explanation.

7.3.3 Contributions

Rubinoff's work in CLEAR provides two major contributions to EF research and development. The first is readily apparent and is concerned with the increased understanding and formal definitions of the types of knowledge contained in a rule. The second is a little more subtle, but was identified in the overview to this chapter. CLEAR does not require that special knowledge types or processing strategies be added to a system in order to work. It simply analyzes the system's existing rules and uses the knowledge it finds there to provide its clarification type explanations.

7.3.4 Conclusion

Rubinoff's work on CLEAR is noted for its revelations concerning the information content of rules. While its particular implementation is dependent on the HPRL language used to develop it, the definitions presented and the methodology described are implementation independent and therefore applicable to a wide range of applications.

7.4 JOE

7.4.1 Background

JOE was developed by Michael Wick and James Slagle [31] at the University of Minnesota in order to provide a more unified and expressive explanation capability for what they identified as **practice** systems (non-research type ESs and ES shells that are being used commercially today). Their work was motivated by the recognition that the HOW/WHY questions that MYCIN type systems can explain from their execution traces/history trees are not extremely expressive, while the current work being done in explanation generation (XPLAIN, Generic Tasks, etc.) involves systems that are too complicated and domain specific to be integrated into common use. More specifically, they noted that the work being done in XPLAIN, NEOMYCIN, and the **deep** knowledge and generic tasks at Ohio State all required the use of supplementary knowledge for their explanations, yet most practice systems are not equipped to provide this supplementary knowledge. Further more, they noted that these systems lack the equipment required to carry on an interactive explanation with the user (natural language processors, English generators, etc.) and are normally unable to model the user's knowledge or intention.

7.4.2 A Journalistic Approach

In searching for a solution to this problem, Wick and Slagle ventured into the area of journalism. Discovering the four **musts** of a newswriter's task to be *accuracy*, *attribution*, *fairness*, and *objectivity*, Wick and Slagle quickly recognized the similarities that existed between this discipline and the functionality of EFs on current practice systems. Both involved presenting readers/users with accurate, objective, noninteractive accounts of events. This being the case, Wick and Slagle pursued their journalistic research further and ultimately formulated the design of their ex-

planation facility from their findings. Surprisingly, the foundation of their design is based on some very familiar principles introduced by a man named Shuman in 1903. These principles form the foundation of news reporting and are as relevant today as they were eighty some years ago. They are called *the five Ws*.

Wick and Slagle based their design of JOE on these five Ws (*Who*, *What*, *Where*, *When*, and *Why*) and a sixth principle - *How* (which, for information purposes was added to the list of five Ws later). Their basic goal was to design a system that would extend the two basic HOW/WHY capabilities of current practice systems to the six just identified. They realized that the explanations this system would produce could only deal with basic, surface level information but were convinced that this type of information could still provide highly informative explanations. The final result of their work can provide eighteen different explanations — the six basic questions, each answered in past, present, and future tense.

7.4.3 Contributions

JOE is a relatively new system and definitely a new concept in explanation generation. How significant this work will turn out to be is yet unknown. However, one feature used in JOE is quite interesting. This feature has to do with the use of time stamps to determine relational links between data values without having this information explicitly stored in the knowledge base. Different time stamps are used for different types of information based on the origin of that information. For example, *user entered values* receive a **time stamp from the system clock** at the time they are entered into the system, *default values* receive a **boot-time stamp** which is set at the beginning of the ES execution, and *inferred values* receive the **largest time stamp** of the data from which they are inferred. Based on this information, Wick and Slagle developed several equations that compared the time stamp values and determined if the relational link between two pieces of data was inferred, default, or

volunteered.

7.4.4 Conclusion

The work done in JOE has approached the issue of explanation generation from a more pragmatic viewpoint than any of the other systems we have examined. Discovering new and exciting ways of producing high level explanations was not the goal of this work, yet enhancing explanation generation definitely was. Identifying the relationship between current EFs and newspaper reporting is quite interesting and provides a mechanism for enhancing many of the explanation capabilities of existing ESs and ES shells.

7.5 Summary

Each of the systems examined in this chapter have made specific contributions to EF research and development. BLAH has provided significant advances in providing Function 2 (and some Function 3) capabilities with its use of the user view knowledge base and its various explanation restructuring techniques. CLEAR has provided invaluable information concerning the knowledge content of rules and has identified a domain-independent methodology for gaining access to this information. JOE has provided the means of enhancing existing, commerically used ES explanation facilities without requiring a major rewrite of the system by recognizing the similarities between the task of the newspaper writer and the functionality of existing EFs using the well established concepts and ideas from the journalism profession.

Chapter 8

Wrapping It Up

8.1 Overview

The purpose of this final chapter in our review of explanation generation research and development is to provide a summary/conclusion to this topic. This will be done using some of the ideas and concepts developed by Roger Schank and his group of researchers at the Yale University AI Laboratory. After briefly discussing the underlying philosophy behind this work we will use several of its key points as the basis for summarizing **where we have been** and identifying **where we are going** in explanation generation research and development.

8.2 A New Focus

Schank, in his book Explanation Patterns: Understanding Mechanically and Creatively [27], identifies an interesting new focus for explanation generation. As his title indicates, this focus deals with the issue of machine understanding and what it means for a machine to do this, a topic that is one of the most fundamental and controversial issues in AI. Schank explains that to understand

> "is to satisfy some basic desire to make sense of what one is processing, to learn from what one has processed, and to formulate new desires about what one wants to learn." [27, page 19]

He identifies three factors that he says "are inextricably bound together" in making up this definition: *explanation*, *learning*, and *creativity*. Furthermore, he notes that

> "Explaining the world, to others and to oneself, is, in essence, the heart of understanding." [27, page 16]

In relating explanation to understanding Schank defines something he calls the *understanding spectrum*. This spectrum provides a measure or gage for determining just how strongly (or weakly) related any one explanation is to understanding — how well it understands. There are three markers along this spectrum that Schank identifies.

1. *Making Sense*: This type of understanding is at the lowest end of the spectrum and represents understanding at the level of being able to identify what sequences of actions an ES has followed during its reasoning process.

2. *Cognitive Understanding*: This type of understanding falls in the middle of the spectrum and represents understanding at the level of being able to explain why an ES came to the conclusions it did, what hypotheses it rejected and why, how previous experiences influenced it to come up with its hypothesis, etc.

3. *Complete Empathy*: This type of understanding is at the highest end of the spectrum and represents understanding at the level of an ES being able to satisfy an examiner or inquirer in the same way that a human would (if that human were in a similar situation).

It is important to point out that Complete Empathy is the category or degree of understanding Schank feels is the cause of all of the controversy with respect to AI and it capabilities. He goes on to note that he does not believe machines will ever reach this level of understanding (provide this type of explanation) in its fullest

sense. This is because complete empathy requires two individuals to experience the same experiences and react to them in the same (or very similar) ways and Schank doesn't believe machines will ever experience enough of these experiences or react to them in the **proper** way(s) to satisfy you or I that it really understands us. As Schank points out, we humans are extremely lucky if we find just one person during our entire lifetime that satisfies the criterion for complete empathy.

This point aside, it is this understanding spectrum and the important relationship between explanation and understanding that we want to use as the basis for our review and preview of EF research and development.

8.3 Where We Have Been

8.3.1 The User

In identifying explanation as the **essence** of understanding, Schank lists two audiences toward which these explanations are to be directed: *others* and *oneself*. In looking at where we have been with respect to the previous chapters of this paper, the first important point to note is that the audience toward which all of this work has been directed is what Schank calls the *others*, people outside the system interested in understanding how and why the system works the way it does. While this may not seem surprising, some interesting points and questions can be raised concerning the specifics of this audience.

First, the majority of the work we have examined in this paper has concerned itself with medical consultation type systems. The entire MYCIN family, Digitalis and XPLAIN, and much of Chandrasekaran's work on generic tasks dealt with these types of systems. This being the case, a valid question concerning the validity and relevance of this work beyond its specific application domain can be raised. Not all ESs are medical and/or consultation systems. The military uses numerous ESs that

are embedded components of larger weapons and communications systems. These systems don't interact with a user. Industry also has numerous non-interactive applications for ESs in such areas as process control and robotics. Why do these systems need an explanation facility when there's no one to explain anything to? The answer, of course, was first identified in Chapter 2 and further discussed in XPLAIN and EES. EFs are an extremely important software engineering tool and since every ES must go through a software development cycle, EFs will always be an important part of any ES. However, using EFs in non-consultative type systems as software engineering tools identifies a second interesting point.

ESs that are embedded in large systems need the debugging, testing, maintenance, and documentation features that EFs can provide. In addition, they require some kind of user interface for use by the system developers. However, once the system has been finished and is ready to be used operationally (ready to be embedded into its larger system), the EF is no longer needed. This implies the need for a modular approach to EF development, creating an EF that can be attached to a system when its software engineering capabilities are needed and then detached from the system before it goes online. However, this apparent requirement creates a major conflict. Throughout the literature it is emphasized that **explanation generation capabilities can't simply be added to an existing ES, but must be a major consideration in the design and development of an ES from the very beginning**. If the types of knowledge an EF needs are intricately woven into the design of the ES, how can the EF be attached and detached on an as-needed basis? One possible answer, I think, can be found in Chandrasekaran's generic task approach. If explanation generation is viewed as a natural capability implicitly provided in the structure of the specific task to be performed instead of as a separate, specific requirement of an ES (with its own special types of knowledge and process

ing schemes), then the only module that needs to be attached and detached is the one that provides the interface to the user.

8.3.2 The Understanding Spectrum

In viewing our previous discussions (Chapters 3–7) with respect to Schank's *understanding spectrum*, we find that a natural progression from the most elementary form of understanding toward a more robust form has taken place. As Schank notes (and is obvious from our discussion of MYCIN) this system provides explanations that fit into the understanding spectrum at the Making Sense level. Using the trace of its execution contained in its history tree, MYCIN is able to explain the sequence of actions it performed in its reasoning process.

While the rest of the MYCIN family also contributed towards improved understanding, the significance of these contributions varied. TEIRESIAS's information metric did help increase the understanding level of the user by presenting various degrees of explanation sophistication. However, the real contributions in understanding came from the work done in GUIDON and NEOMYCIN. While their work on explaining control strategies falls short of providing complete Cognitive Understanding, it definitely improves MYCIN's initial understanding level.

Swartout's work in Digitalis improves only slightly on MYCIN's initial understanding level and would therefore fall slightly to the right of MYCIN on the understanding spectrum. This slight improvement of course, is contained in the procedural approach that Swartout used and the ability the system had to explain these procedures. However, significant improvements were made in XPLAIN. While still falling short of complete Cognitive Understanding (as Schank defines it), the automatic programming concept used in this system captures much of the **missing** programmer information needed for better explanations (explanations that are more understandable). EES further extends these improvements, coming even closer to

fitting Schank's definition of Cognitive Understanding.

Chandrasekaran's work with generic tasks is difficult to asses. The overall concept definitely appears to provide the capabilities necessary to produce explanations on the level of Cognitive Understanding. However, as no real analysis of the effectiveness of this concept has been provided, we can only speculate.

Each of the systems discussed in chapter 7 were designed to improve various aspects of an ES's explanation generation capabilities. From this standpoint, each was concerned with helping improve the level of understanding provided by these EFs. However, I think it is safe to say that each of these systems would fall just to the right of the Making Sense level on the spectrum.

8.4 Where We Are Going

As already noted, all of our previous discussions have dealt with systems whose EFs were designed with some outside user in mind. Whether this user is a *domain* expert or system developer is not important. What is important is that in looking at one direction of where we are going in future EF research and development, we must change our focus of attention and concentrate on the second audience Schank identified, that being *oneself* (which in our case refers to the system itself).

As mentioned earlier, Schank believes explanation, learning, and creativity are wrapped up together in defining *understanding*. This is based on his observation that understanding is a dynamic entity that changes over time and that people adapt to new circumstances by changing their behavior. Schank argues that we do this through *self-explanation*. He points out that when we explain things to ourselves we are attempting to correct some initial misunderstanding by finding relevant experiences we have had that might account for the misunderstood event or situation. When we do this we obtain a new understanding, creating a new

explanation of this understanding as a result. Schank argues that the creativity involved in coming up with this new explanation is the essence of what it means to think and therefore the heart of what we mean by understanding. He also argues that this creativity is not some magical gift we humans possess, but rather is a mechanistic process that is possible for computers to replicate. Work is currently being conducted in developing such a process and is the major emphasis of Schank's book [27]. While this process is beyond the scope of this paper, the important point to note is that one future direction of EF research is in the area of *self-directed explanations* for the purpose of developing machines that are **creative**.

Interestingly, creativity is not the only motivation behind self-directed explanations. Hammond [19] has been working on a planning system that uses causal explanations to repair plans that initially fail. Plan failures are described in terms of causal explanations of why they occurred and then used to access different abstract planning strategies, which , once an appropriate strategy is found, results in a specific change being made to the faulty plan. The important point to recognize here is that the causal explanations generated for use by the system are the exact same explanations that would be presented to an inquisitive user. They are not some special new form of explanation.

A higher degree of understanding is the goal of EF research and development. Not surprising is the fact that this focus on understanding is fundamental to the definition of explanation we established in Chapter 2. However, presenting these explanations to the system itself, as well as to an inquisitive outside user, appears to be the direction of future EF research and development.

Chapter 9

EFFESS: Background Information

9.1 Overview

As identified in the introduction to this thesis, this work on *explanation generation* in ESs is being conducted from two different points of view and is part of a larger research and development effort being conducted at Wright State University into the use of two important (and somewhat conflicting) technologies: *software engineering with Ada* and *AI applications*. The purpose of this chapter is to provide some important background information concerning the second aspect of this thesis, the implementation of EFFESS. Specifically, this chapter will first describe the ES shell used in this project. It will then discuss the initial requirements of this project and re-examine the literature review in order to identify the concepts and ideas that have influenced it. Additionally, information taken from sources not presented in the literature review will also be discussed.

9.2 The ES Shell

The ES shell being used in EFFESS was developed by Capt James Cardow as part of his master's degree requirements [5] and represents one effort resulting from the *Ada and AI* work being done at Wright State University. In order to understand

much of what will soon be discussed with respect to EFFESS, a basic description of this ES shell is needed.

Cardow's shell was designed using the Object Oriented Programming (OOP) design methodology, partly because Ada supports this methodology and partly because OOP is closely related to the *frame-based* KRS Cardow chose to use. Reasoning within the system is performed by using either a *forward-chaining (data driven)* process, a *backward-chaining (hypothesis driven)* process, or a combination of both of these processes. Information in the system is stored in a *hierarchy of frames* and can be passed between these frames by using *inheritance* or by passing *messages*. *Demons* are attached to the slots of the frames and represent the execution mechanism of the system. Each type of demon performs a specific function on the slot to which it is attached and accomplishes this function by processing or **firing** an associated set of *production rule(s)*.

While this description is by no means complete (the interested reader is referred to [5] for complete details of Cardow's work), it does identify the key features of the ES shell. The fundamental goal of EFFESS is to explain these different features to a user of the system and to do so in an understandable way.

9.3 Initial Requirements

There are two major requirements for EFFESS that need to be discussed as part of the background information for this project. These are: the use of Ada as the implementation language for the project and the decision to **add** an EF to an existing system rather than **include** an EF in the design/redesign of an ES. The reason it is important to understand these requirements is because they define the *scope* of the EFFESS project.

The need to include sound software engineering practices in the development

of **any** computer software system is a well accepted fact. One important aspect of doing this is to use implementation languages that explicitly provide software engineering constructs as part of the language. Ada was specifically designed to be one of these languages. However, the extremely flexible, rapid prototyping languages specifically designed for AI research and development (LISP and PROLOG) were not designed with software engineering support in mind. While their use in an academic or research laboratory type setting has been accepted, as more and more of AI's technology (i.e. expert system technology) moves out of this setting and into an operational, on-line environment, the use of software engineering intensive languages to develop these AI systems must be investigated. Thus, the use of Ada as the implementation language for EFFESS is one requirement.

The requirement to **add** an EF rather than **include** its requirements and specifications in the design or redesign of an ES developed from three different sources. The first, which was initially identified in Chapter 8, deals with the need established for a *plug-in* type EF that provides a testing and debugging type explanation capability for ESs that are ultimately going to be embedded into larger systems of some kind. The second, which was initially identified in Chapter 7 during the discussion of Wick and Slagle's work on JOE [31], deals with the issue of *functionality versus cost*. Wick and Slagle recognized that many operational ESs (ones they called *practice systems*) needed an EF but could not afford to go through a major redesign in order to acquire some of the sophisticated explanation capabilities we have examined in previous chapters. They also recognized that an *effective* EF could be provided without going through this redesign process. Therefore, their effort was concentrated on providing effective EFs for on-line ESs **without making major modifications to the original ES code**, an effort that was directly applicable to this project. The third contributing source to this requirement was the *frame-*

based KRS Cardow chose to use in the ES shell. After examining Cardow's shell, definite support existed for viewing this requirement as a feasible one. (Chapter 10 discusses this frame-based KRS in detail and identifies the contributions it makes to explanation generation.)

9.4 The Literature Review Revisited

As identified in the introduction to this thesis, the purpose of the literature review was to establish a knowledge base to draw from in determining the functionality and development of EFFESS. As it turned out, this knowledge base is quite large and includes a wide range of topics related to explanation generation in ESs, not all of which are applicable to this effort. This section identifies those concepts and ideas that have been taken from this knowledge base and used (in one way or another) in EFFESS.

- *MYCIN* (Chapter 3) : The basic functionality of MYCIN's Reasoning Status Checker (RSC) is used in EFFESS. In addition, the two example questions listed for the RSC are implemented. While the functionality of the GQA was considered, its requirement for natural language processing was determined to be beyond the scope of this project. However, access to the different types of information required by the GQA is made available in EFFESS.

- *The MYCIN Family* (Chapter 4)

  1. *TEIRESIAS* : The four step process Davis identifies as being necessary to design an EF was used. Chapter 11 presents a discussion of EFFESS' design process using these four steps.

  2. *GUIDON* : While specific use of *meta-rules* was not needed in EFFESS, *meta-knowledge* concerning the implicit inheritance control mechanism of

the frame-based KRS was used.

3. *NEOMYCIN* : The primary focus of NEOMYCIN was on explaining diagnostic strategies. While EFFESS does provide a limited capability in this area, little of the work done in NEOMYCIN was used.

- *Explicit Development Models* (Chapter 5) : Most of Swartout's work involves the capturing and explanation of *deep* knowledge. While his *automatic program writer* is an interesting concept, providing this type of functionality is beyond the scope of this project. However, two aspects of EFFESS can be traced back to Swartout's work. First, the production rules in the system were encoded using descriptive, English-like names. This is a powerful feature of Ada and allows EFFESS to produce very descriptive explanations without having to concern itself with English translations of rules, text generators, etc. Swartout used a similar capability in The Digitalis Therapy Advisor. Second, Swartout recognized the software engineering assistance EFs can provide. As has already been identified, creating a *plug-in* software engineering tool is one of the requirements/objectives of this project.

- *Generic Tasks* (Chapter 6) : Chandrasekaran's development of specific design languages in support of his generic tasks is beyond the scope of this project. However, an interesting observation concerning this project and these generic tasks is presented in Chapter 12.

- *A Potpourri of EFs* (Chapter 7)

1. *BLAH* : None of Weiner's work in BLAH is used in EFFESS because major changes to the ES shell would be required. However, a discussion of his *system view* and his *user view* with respect to EFFESS is presented in Chapter 12.

2. *CLEAR* : Rubinoff's CLEAR system was designed to be an attachable front-end to an **independently developed** ES. While none of the functionality of CLEAR is included in EFFESS, the basic design goal of CLEAR is one of the primary objectives of this project.

3. JOE : Wick and Slagle's work on JOE has been, by far, the most influential on this project. The applicability of their basic philosophy was identified earlier in this chapter. In addition, four of the six functions they provide in JOE are implemented in EFFESS: WHAT, WHERE, WHY, and HOW. (However, these four functions are not presented in three different tenses, as was done in JOE.)

While we are discussing the literature information that has influenced the development of EFFESS, one article not presented in the literature review needs to be mentioned. In one section of their article, Building Knowledge-Based Systems with Procedural Languages [3], Butler, Hodil, and Richardson discuss EFs. In their discussion they identify three functions that an EF should perform: the Rule Query, the Why Query, and the Explain (or How) Query. Additionally, they identify the use of a *stack* as an appropriate data structure for maintaining a trace of the system's performance. All of this information is included in the functionality of EFFESS. However, the implementation of the Why Query and the How Query vary from the descriptions Butler, Hodil, and Richardson provide for these functions. EFFESS's ... and Why functions are based on MYCIN's interpretation of what these queries mean. The reason for this difference stems from the fact that both systems are avoiding the problems of natural language processing by explicitly defining what is meant by Why and How. As these two words have several different meanings (hence the problems associated with natural language processing), MYCIN has defined the functionality of these queries differently than Butler, Hodil, and Richardson. We

have chosen to implement MYCIN's version.

9.5 Summary

This chapter has presented important background information concerning the ES shell that is to be explained, the initial requirements established for the project, and the concepts and ideas that have been taken from the literature on explanation generation and used in the development of EFFESS.

Chapter 10

EFFESS: A Frame-Based System

10.1 Overview

One of the most important aspects of our project is that it involves the explanation of a *frame-based* KRS. Therefore, to understand the specifics of this implementation effort, a detailed examination of this KRS is required. The purpose of this chapter is to provide this detailed examination. Interestingly, there are several differing opinions within the AI community concerning the definition of a *frame-based* knowledge representation structure or *frame-based* system. While understanding these differences is not important for our discussion, understanding the two fundamental KRSs upon which they are based is important. Therefore, this chapter begins by discussing these two KRSs (rule-based and frames) individually, as the basis for understanding the KRS used in EFFESS. Once this understanding has been established, a description of our *frame-based* KRS will be presented, followed by a discussion of the contributions it makes to explanation generation.

10.2 A Rule-Based KRS

10.2.1 Background

In his chapter on *Production Systems* (which is another name for a Rule-Based ES) [17, chapter 10], Firebaugh notes that the concept of a *production* system or a *rule-*

based system was first introduced by Emil Post in 1943. He goes on to say that this type of system was proposed as a model of human problem-solving behavior (the context in which it is used in AI) by Allan Newell and Herbert Simon in 1973. This problem-solving model and KRS have had, and continue to have, a significant impact on ES development. Much of the work described in chapters 3 - 7 of this thesis are production systems, using rule-based KRSs.

10.2.2 Composition

There are three main components of a rule-based ES.

1. *Knowledge Base* (KB): The KB consists of a set of *if — then* or *condition — action* rules. (i.e. *if* condition *then* action) The expert knowledge for the system is encoded in these rules.

2. *Control System* : Also known as the *inference engine*, this part of the system is the rule interpreter and rule orderer/sequencer. The control system provides the processing direction and operation needed to run the ES.

3. *DataBase* : The database is the storage facility that contains the information the *rule conditions* evaluate and upon which the *rule actions* act.

10.2.3 Strengths

Firebaugh [17] identifies four major strengths that have made rule-based systems so popular. These are:

1. *Natural Representation* : A rule-based KRS provides a natural way to represent the " *if such and such, then do such and such* " type situations often encountered by human experts.

2. *Simple Construct* : A rule-based KRS is a simple, uniform way to represent knowledge. The *if — then* construct of the production rule is easy to

implement, is often times self-documenting, helps simplify the communication between different parts of the system, and aids in providing English-like definitions of the rules (a factor that MYCIN exploited in producing its English-like explanations).

3. *Modular and Modifiable* : Each rule in a rule-based KRS represents a discrete piece of knowledge and is usually not directly related to any other rule in the KB. Therefore, *information* can be viewed as a collection of independent facts, each of which can be easily modified.

4. *Knowledge Intensive* : The KB in a rule-based KRS contains **all** of the expert knowledge for the system and **only** this information. It is knowledge intensive. Therefore, it is separate from the rest of the system. This allows the reuse of a particular control strategy since it is not dependent on the KB at all.

10.2.4 Weaknesses

Firebaugh [17] identifies two weaknesses or limitations of production systems that are worth mentioning. The first has to do with the fact that, while individual rules are clear and concise and easily understood, when they are combined into the overall system this clearness and conciseness tends to disappear. The second has to do with the lack of structure or organization that exists for systems containing large numbers of rules. Usually, all of the rules are contained in one large list or set. This requires an exhaustive search of this list or set each time a rule is needed. Obviously this is not very efficient. While these two weaknesses are significant, the real problem with rule-based systems (from an explanation generation point of view) is identified by Fikes and Kehler in their article entitled The Role of Frame-Based Representation in Reasoning [16, page 907]. Simply stated, Fikes and Kehler point out that rule-based systems can't: *define terms*,

describe domain objects, or *identify static relationships among objects.*

10.3 A Frame KRS

10.3.1 Background

This concept was originally introduced by Marvin Minsky in 1975 as a result of his work on scene analysis and computer vision.[22] Minsky's concept was based on the idea that there are a number of *stereotypical* situations that we encounter on a regular basis and that these situations can be represented in a hierarchically structured framework. For example, a *stereotypical room* is made up of four walls, a floor, and a ceiling. Therefore, the top level of the hierarchical structure would be the *room* object. The next level of the hierarchy would contain six objects: one for each of the four walls, one for the floor, and one for the ceiling. The third level of the hierarchy would consist of the objects related to a particular object in the second level. For example, a wall may have a door or window in it, a picture hanging on it, etc. Therefore, the third level of the hierarchy would contain descriptions of these objects, relating them back to the higher level object of which they are a part. This structural breakdown would continue until the *stereotypical room* was adequately represented.

10.3.2 Composition

The primary data structure of a frame-based KRS is the *frame*. A frame describes an object or a class of objects and contains *slots* that are used to identify or describe whatever is being represented by the frame. The slots of a frame contain one of two types of values. They either contain default or constant values that describe the object being represented by the frame or they contain variable values that can be changed, based on the values of other slots of other frames in the hierarchy.

Additionally, frames can have sub-frames attached or related to them. It is this frame — sub-frame structure that makes up the hierarchy of this KRS.

Another feature included in this KRS is an implicit control structure resulting from the hierarchical relationships of the structure. This control structure is commonly referred to as *inheritance*.

10.3.3 Strengths

Firebaugh [17] identifies several strengths associated with a frame-based KRS.

1. A number of different objects may share the same frame. Referring to the *stereotypical room* example above, if we wanted to describe several different rooms in a building, each of these rooms could share the same basic structure of our *stereotypical room* because all rooms (or at least, most rooms) have four walls, a floor, and a ceiling. The slot or attribute values of these frames would be different for each type of room, but if the basic structure were general enough it could be used to represent any room.

2. As already mentioned, frame-based KRSs provide a natural hierarchy for structuring knowledge which often times captures the way an expert thinks of or describes some object.

3. This structure also provides a concise representation of the relationships that exist between different objects, which in turn provides for easy support of information queries (simply find the frame and search its slots for the desired information).

4. This type of KRS also provides an easy mechanism for defining default values that can be overwritten at a later time, should the need arise.

10.3.4 Weaknesses

While some criticism has been raised concerning a frame-based KRS's ability to handle *atypical* situations [17, page 293], the major weakness of this type of KRS is again identified by Fikes and Kehler. While frame-based systems provide a rich knowledge representation structure, they don't have a direct facility for "declaratively describing how the knowledge stored in the frames is to be used". [16, page 905]

10.4 A Frame-Based KRS

As can be seen from the previous two discussions, each of these KRSs has a major limitation with respect to explanation generation (as noted by Fikes and Kehler). It should also be fairly clear that by combining these two KRSs, the strengths of one overcome the limitations of the other. Rule-based systems can't define terms, describe objects, or identify static relationships among objects. Frame-based systems can do this quite effectively. On the other hand, frame-based systems can't declaratively describe how to process the knowledge they contain. Rule-based systems do this extremely well. Therefore, by combining these two KRSs a much more powerful and robust KRS is created. Fikes and Kehler provide an excellent description of this new KRS [16] and it is this KRS that is actually used by Cardow in his ES shell.

10.5 Contributions to Explanation Generation

There are several contributions that this *hybrid* KRS makes to explanation generation.

1. Because the production rules are attached to the slots via demons, a logical partitioning of the rule set is provided. Thus, a partial explanation as to

the purpose of a given rule can be provided simply by examining the slot information to which it is attached.

2. The hierarchical structure identifies the relationships among objects and allows for an explanatory description of an object by simply identifying the sub-frames attached to it. Further explanation of each of the sub-frames can also be provided by identifying the slots attached to each sub-frame.

3. The implicit control/inferencing mechanism (*inheritance*) in this system is available for explanation and thus provides some Function 1, Type 3 explanation capabilities. In addition, slot values that were determined by inheritance can also be identified. This additional information enhances the explanation of that particular slot.

4. The hierarchical structure also provides a logical partitioning of the knowledge. Therefore, if an EF were to be enhanced by providing **deep** descriptive knowledge for an object, this information could be easily added as a slot to the appropriate frame or as an attribute to the appropriate slot.

5. As already noted by Firebaugh, this type of KRS provides an easy, efficient process for handling queries about the KB because the knowledge is so well structured and easy to find.

6. This KRS provides two *typical* representations of expert knowledge (if — then rules, and object decomposition) that map almost directly to the way this knowledge is thought of by experts in the real world. Therefore, explanations of these objects should provide more understandable explanations simply because of their realistic representations in the system.

10.6 Summary

By presenting detailed descriptions of two fundamental KRSs used in AI and by identifying their strengths and weakness, this chapter has identified a powerful new KRS created by combining these two KRSs, identified several contributions this new KRS provides with respect to explanation generation, and emphasized that this KRS is the frame-based KRS used by EFFESS.

Chapter 11

EFFESS : Design & Functionality

11.1 Overview

The chapter discusses the design process used in developing EFFESS and describes the specific functions EFFESS performs. It then looks at the functionality of this system with respect to the Functional Framework presented in Chapter 2.

11.2 The Design

In deciding how to go about designing EFFESS, two concepts or ideas were used. The first was Davis' four step design process for EFs presented in Chapter 4's discussion of TEIRESIAS. The second was the OOP methodology used by Cardow in designing the original ES shell.

11.2.1 Davis' Four Step Design Process

Repeating the information presented in Chapter 4, the four steps Davis describes for designing an EF are:

1. Determine the program operation that is to be viewed as primitive.

2. Augment the program code to leave a trace or record of the system's behavior at the level of detail chosen in (1).

3. Select a global framework in which the trace or record can be understood.

4. Design a program that can explain the trace or record to the user.

These steps will now be discussed with respect to how they were applied in the design of EFFESS.

The primitive operation chosen for EFFESS is the same as the one Davis chose for TEIRESIAS, the *invocation* or *firing* of a rule. While the execution of the *demons* in our frame-based system is identified as providing the execution control of the system, demons are not the most primitive operation. The specific operation a particular demon is identified to perform is carried out by firing the set of rule(s) associated with that demon. Therefore, the individual production rule is the primitive operation in EFFESS.

The execution trace in EFFESS is a stack data structure. As identified in Chapter 9, this idea was taken from Butler, Hodil, and Richardson's article on using procedural languages to build knowledge-based systems. A stack provides a straightforward mechanism for properly recording the order in which the rules were fired.

The global framework in which the execution trace can be understood in EFFESS is its frame-based KRS. Davis chose a *goal tree* for his work in TEIRESIAS because of the backward-chaining control structure used in MYCIN. While our system also provides a backward-chaining control structure (and a forward-chaining one as well), the framework for understanding the execution trace (for understanding why a rule was tested/fired at a particular time during the execution of the system) involves information contained in the KRS. Remember that information is passed between frames in one of two ways (through inheritance or by passing messages) and that demons are executed when a slot's value is requested but does not exist. Therefore, to understand why a rule was tested/fired at a particular time (as indicated by the execution trace) requires that we know the method that was used in attempting to

obtain a value for the slot. This information is contained in the KRS.

This final step of Davis's EF design process (writing a program to explain the trace to a user) has been expanded in EFFESS. This is because several of the explanation functions of EFFESS don't use the execution trace in providing their explanations. They are able to get the information they need directly from the KRS. Therefore, the EFFESS program (which is contained in **one** Ada package and is presented in the appendices) contains more than just the code related to explaining the execution trace to the user.

11.2.2 OOP

The OOP design methodology presented by Booch [1, Chapter 4] was used in EFFESS' design at three different levels of abstraction. A discussion of its use at the highest level of abstraction (the *complete ES Development Environment* level) is presented here. EFFESS and the ES shell are the two primary objects that have to interact in order to provide the functionality of this larger system. As the ES shell was designed using OOP (before EFFESS existed), the only change that needed to be made was in its visibility. The ES shell has to have access to (be able to see) EFFESS in order to build the execution trace. It also has to have access to EFFESS's processing entry procedure in order to pass control when it is time to generate explanations. Because of Ada's strong support of OOP, providing this visibility for the ES shell was easily done.

With respect to using OOP in designing EFFESS, the operations identified for this object are the explanatory functions it must perform and are described in the next section. The visibility EFFESS requires consists primarily of the ES shell's processing package and its KRS. (Visibility was also established to a useful package of I/O routines. However, this was done from a code reusability standpoint and was not functionally required). As for establishing EFFESS's external interface, the

routines it uses to build its execution trace and its processing entry procedures have to be made available to the ES shell.

It is important to note that while the ES shell and EFFESS are the only two objects identified for this *complete ES Development Environment*, as other objects are developed (i.e. a user interface, a natural language processor, etc.), they too can be easily incorporated into the system using this same procedure.

11.3 Functionality

In looking at the functional requirements of EFFESS, two different explanation environments were identified: the *runtime* environment and the *end-of-processing* environment. Interestingly, the explanation requirements for these two environments are not the same. While they do share several of the same requirements, each has some unique requirements as well. For example, during the *runtime* environment the user may be prompted for information needed by the system. Being able to explain WHY the system needs this information is an important explanation requirement in these situations. However, the *end-of-processing* environment has no reason to prompt the user for information, therefore no requirement exists for a WHY explanation capability. Conversely, once the system has completed its processing and arrived at some kind of decision, being able to SHOW the *critical decision path* (the sequence of rules that fired) the system used to arrive at this decision provides a great deal of important information. However, during system execution, explaining the *current decision* path is the important issue. Therefore, in support of these differences, EFFESS provides a set of *runtime explanation functions* and a set of *end-of-processing explanation functions*.

- Runtime Explanation Functions

1. *Explain Rule* : The Explain Rule function explains the identified rule by displaying its contents to the screen. As descriptive, English-like naming conventions were used in the construction of the rules, no additional text generation is required to present an understandable explanation.

2. *Explain Why* : The Explain Why function is interpreted to mean, **Why is this information being requested?** and is presented to the user as an option whenever the user is prompted for information. The basis for providing this explanation is twofold. First, some component in the system (i.e. a rule) needs this information in order to continue processing. Second, this value is currently unknown and couldn't be determined via inheritance or message passing. Therefore, EFFESS's WHY explanation identifies the slot whose value is being requested, the frame to which this slot is attached, and the system component that is waiting on this slot in order to continue processing.

3. *Explain How* : The Explain How function is interpreted to mean, **How did the system arrive at this point in its processing?** and uses the execution trace to provide this explanation. This function starts with the first rule tested by the system and recurses through the execution trace for as long as the user determines is necessary. One rule from the trace is explained for each successive HOW request the user provides. He can examine the entire execution trace (from the start rule all the way to the current rule being processed) or he can stop at whatever level he is comfortable/satisfied with. The explanation content for each rule is based on the specific type of rule it is. For example, rules associated with GOAL frames are chosen for execution because they represent the specific goal to be achieved. Therefore, EFFESS's HOW function explains these rules

in this context, identifying the rule number, the goal to be achieved, etc. On the other hand, rules that prompt the user for information are chosen for execution because the system needs this information to continue processing. Therefore, these rules are explained with respect to the system component that is dependent on them for the needed information.

4. *Explain What* : The Explain What function is interpreted to mean, **What is the value of slot X?** and is available for use throughout the execution of the system. The user must provide the name of an attribute (slot) and will receive its corresponding value (or a *not found* error message) in return. This function represents one type of query function Firebaugh identified as one of the features easily supported by a frame-based system.

5. *Explain Where* : The Explain Where function is interpreted to mean, **Where is X located in the KRS?** and is used to explain the relationships of objects and attributes in the KB. The user must provide an object name or an attribute name and will receive an explanation of who/what the input item is related too. If the input item is an object, any inheritance relationships to other objects are identified. If the input item is an attribute, the object to which it is attached is identified. This function represents a second type of query function.

- End-Of-Processing Explanation Functions

1. *Explain Rule* : same as in runtime functions
2. *Explain What* : same as in runtime functions
3. *Explain Where* : same as in runtime functions
4. *Show Critical Path* : The execution trace is used to provide this function. As the trace maintains the correct ordering of all rules tested during the

system's execution, the critical path list is provided by simply looping through the execution trace and printing out the rule numbers of the rules whose *fired* flag has been set.

5. *Show Execution Trace* : This function also uses the execution trace to accomplish it function. Its purpose is to display the entire sequence of rules that were tested during the system's execution. However, a rule can be in one of three states during various stages of the system's execution: *pending* (awaiting information), *fired* (its antecedent tested true), or *failed* (its antecedent tested false). Therefore, this function identifies which state a particular rule is in at the various stages of its execution.

11.4 Functional Framework

In analyzing EFFESS with respect to the Functional Framework presented in Chapter 2, we find that Function 1, Type 1 explanations are provided by the HOW function, the SHOW Critical Path function, and the SHOW Execution Trace function. By using a stack to capture the execution trace of the system (the order in which the rules were accessed), the basic decision process of the system is identified. Type 2 explanations are provided by the Rule, What, Where, and Why functions, all of which provide information about the elements (objects and attributes) in the knowledge base. Function 1, Type 3 explanations are also provided to a limited degree. The Why function can identify certain instances when inheritance or message passing has been used.

EFFESS provides limited Function 2 capabilities by allowing the user to determine the degree of explanation provided by the HOW function. However, as the primary users of EFFESS are assumed to be system designers and knowledge engineers, based on the previously discussed need for *plug-in* test and debugging type

EFs), the explanations provided in this system have been directly geared for these users.

With respect to Function 3 capabilities, EFFESS does provide English-like output due to the descriptive naming conventions used in the system. However, considerable room for improvement still exists in this area.

Chapter 12

EFFESS: Conclusions & Future Work

12.1 Overview

This chapter presents some concluding observations and remarks concerning the EFFESS project and considers several possible directions for future work. In so doing, three important issues are discussed: the use of Ada as an implementation language for this AI application, the positive and negative aspects of *attempting* to add an EF to an existing ES, and the major contributions that a frame-based system provides with respect to generating explanations.

12.2 Conclusions

12.2.1 A Straightforward Development Process

The first observation or conclusion about this project is that its development and implementation proved to be a much more straightforward process than originally anticipated. This is due to three major factors: the use of the OOP design methodology, the use of Ada, and the use of a frame-based ES shell.

Using the OOP design methodology provided several contributions to the development process. Probably the most influential is the fact that in analyzing the ES

shell and the desire to add an EF to it (from an OOP point of view), enough of a feasible solution was identified to make the decision to attempt the effort in the first place. Additionally, the required visibility and interface between EFFESS and the ES shell were easily identified using this design methodology.

Closely coupled with the contributions OOP provided to the design of EFFESS were the contributions Ada provided in support of these OOP decisions. Due to the already identified relationship that exists between Ada and OOP, once the visibility and interface specifications had been established, implementing them in Ada was a straightforward process using the built in contructs it specifically provides for these purposes (i.e. *with* clauses, *package specifications*, *separate compilation units*, etc.).

With respect to the frame-based KRS used in the ES shell, many of the contributions this structure provides for explanation generation were described in Chapter 10. These contributions are universal in nature, in that they are provided by a frame-based KRS implemented in any language. However, a frame-based KRS *implemented in Ada provides an additional contribution to explanation generation* as a result of its **strong typing** requirements. While these requirements are being identified as a major contribution to the development of EFFESS, it is interesting to note that these same requirements were identified as a major obstacle in the development of the ES shell (see [5, Chapter 3] for complete details). In any case, Ada's strong typing required the different objects of the ES shell to be decomposed into different structures of nested records and record pointers in order to be implemented. The contribution this makes to EFFESS is that each of the decomposed parts of an object are available for explanation. Therefore, in a frame-based KRS implemented in Ada, a hierarchical decomposition of objects at two different levels of abstraction are available for explanation. At the *knowledge level* this hierarchy is provided by the KRS. At the *implementation level* this hierarchy is provided by the

different structures of nested records and record pointers mentioned above.

12.2.2 A Plug-In & Unplug Type EF

In evaluating how successful we were in accomplishing our *plug-in* EF objective, two answers are required. Overall, the effort appears to be a success. Absolutely none of the original ES shell data structures were changed at all and the processing routines were only changed (added to) in three places: in the routine where the execution trace had to be built, in the routine that interfaced with the user so as to gain access to EFFESS, and in the driver routine to provide the end of processing explanation capabilities. Due to Ada's *separate compilation* construct, all of EFFESS' code is contained in one Ada package. Therefore, to *unplug* EFFESS from the ES shell requires commenting out or removing fifteen lines of code in three different routines (see Appendix C), removing the EF package from the compilation order, and recompiling the ES shell code.

From an explanation standpoint we can also consider EFFESS a success in that effective, useful explanations are provided, especially with regards to Type 1 and Type 2 explanations. However, we can also consider EFFESS to be only marginally successful because of its limited ability to produce Type 3 explanations. While this is true, it is important to note that the reason Type 3 explanations are limited **is not** due to a lack of explainable information but rather because we restricted ourselves to not changing any (or as little as possible) of the ES shell code. Had this restriction not existed, several Type 3 explanations could have been provided because information concerning inheritance, message passing, and the forward/backward chaining control mechanisms is available to be explained.

12.2.3 Cost versus Capability

In many ways, the entire EFFESS project has re-addressed or re-focused our attention on a very old and important issue in the use of any new technology, *cost* versus *capability*. Only finite amounts of resources (time, money, and manpower) exist for any given project. Often times, the cost of incorporating the most *state-of-the-art* capabilities a technology provides exceeds these finite limits. Therefore, efforts at identifying ways to provide **the most capability for the least amount of resources** are not only justified, but severely needed. This project has been one of these efforts. However, it is important to realize that a frame-based ES provides the potential for producing a much richer explanation capability than provided in this project, if one includes the requirements and specifications of an EF in the initial design stages of the ES. The following section identifies some of these capabilities.

12.3 Future Work

One of the most significant discoveries of the EFFESS project was that a frame-based KRS is a rich structure for generating explanations. This section identifies several areas of future work intended to exploit this fact.

12.3.1 KRS Expansion

We have already pointed out that virtually everything in a frame-based system is available for explanation. Therefore, if the frame-based KRS is expanded to include more information, the EF that uses this KRS will be able to provide expanded explanations. Fikes and Kehler [16] provide an excellent description of a frame-based KRS and how it can be modified to provide different capabilities. They describe a system that is made up of frames, slots, facets, and subfacets, each of which is an attribute of the preceding object. By using these different levels of attributes, as

much or as little constraining/descriptive information can be appropriately related to any item in the KB. For example, one could add facets to a slot that identifies the maximum and/or minimum values allowed for that slot. Confidence factors identifying the degree of certainty or reliability of a slot's value could also be attached using a facet. If one were to implement the rules in the system as frames (something Fikes and Kehler discuss), considerable information could be identified and stored for each rule. Historical information about the date and author of a rule could easily be stored. So could English translations of the rule, descriptions as to the rules importance and use, reasons why a rule failed to fire, etc. As can easily be seen, including some (or all) of these pieces of information would definitely improve the EF of the system, as all of them would be available for explanation.

12.3.2 User View

Our initial effort in EFFESS didn't address Function 2 issues to any significant degree. However, the support a frame-based system could provide in creating a *User View*, similar to the one Weiner developed in BLAH, could be seen as EFFESS developed. For example, one could add a *User Identification Slot* to each frame in the system, indicating the degree of knowledge the user had concerning that particular object, and then provide different levels of explanation of that object based on the user's knowledge level. Weiner's creation of two separate views of the KB could also be supported by a frame-based system, using the *object relationship identification capabilities* this type of system provides.

12.3.3 Graphical Display of KRS

Because of the natural identification of relationships among objects provided in a frame-based system, being able to display these relationships in graphical form would provide a tremendous explanation capability. The old cliche that "a picture

is worth a thousand words" is especially true when one is confined to displaying information on a twenty-five line video screen. Being limited to the use of textual descriptions and needing to keep related pieces of information on the screen long enough to identify the importance of their relationship, is very difficult to do. Drawing pictures to show these relationships is a much more effective way of presenting the information.

12.3.4 Beginnings of a Generic Task

Repeated emphasis has been given to the rich explanation generation support a frame-based system provides. Interestingly, if we step back and look at the total system that is produced by EFFESS (the ES shell and its explanation facility), we can see the definite beginnings of an environment very similar to one of Chandrasekaran's generic tasks. Remembering that these generic tasks are intended to provide a framework that unifies three major areas in ESs: the *problem-solving process*, *deep cognitive models*, and *explanation*, we can see this basic framework in EFFESS. We have discussed each of these three areas with respect to a frame-based system during our previous discussions, but have not tied them all together or related them to Chandrasekaran's work. Our system provides a problem-solving process (a set of control strategies) with its forward and backward chaining, inheritance, and message passing. While our system doesn't presently contain any **deep** knowledge, the mechanism for adding it to the system has been identified and is a straightforward process for a frame-based system. Our system has a basic explanation facility, and while it is currently a limited one, we have identified the fact that everything in a frame-based KRS is available for explanation and therefore expanding our EF into a more robust system would not be difficult. The only thing that is needed is a development language to tie this all together.

Appendix A

EFFESS Code

```
--|*********************************************************************
--|*********************************************************************
--|*********************************************************************
--|**********                                                 **********
--|**********   EXPLANATION   FACILITY   SPECIFICATION   **********
--|**********                                                 **********
--|*********************************************************************
--|*********************************************************************
--|*********************************************************************

with Frames, Rules;

package Ef is

--| Procedure Name:  CLEAR_EXECUTION_TRACE
--| Inputs: None
--| Outputs: None
--| Purpose: To clear or re-initialize the execution trace.
--| Notes: The ES shell provides a menu of processing options upon
--|   completion of its initial execution.  Two of these options
--|   allow the user to re-run the shell with the same knowledge
--|   base file or with a different knowledge base file.  If either
--|   of these options are selected, the execution trace must be re-
--|   initialized. Called by ES shell driver program, Procedure
--|   SHELL.

  procedure Clear_Execution_Trace;



--| Procedure Name:  ACCESS_RUNTIME_EF
--| Inputs: None
--| Outputs: None
--| Purpose: Provide access to explanation facility during execution
--|   of the ES shell (during runtime).
--| Notes: Called by Function GET_USER_PROMPT in Package USER_IO of
```

```
--|   ES shell.

  procedure Access_Runtime_Ef;


--| Procedure Name:  ACCESS_END_OF_PROCESSING_EF
--| Inputs: None
--| Outputs: None
--| Purpose: Provide access to explanation facility after the ES
--|   shell has completed execution.
--| Notes: Called by ES shell driver program, Procedure SHELL.

  procedure Access_End_Of_Processing_Ef;


--| Procedure Name:  BUILD_EXECUTION_TRACE
--| Inputs: The current frame being accessed by the ES shell.
--|         The current rule being tested by the ES shell.
--| Outputs: None
--| Purpose: Builds the trace of the ES shell execution that is used
--|   by the explanation facility to generate many if its
--|   explanations.
--| Notes: Called by Procedure FIRE_RULES in Package DEMONS of the
--|   ES shell.

procedure Build_Execution_Trace (The_Frame : in Frames.Frame_Ptr;
                                 The_Rule : in Rules.Rule_Ptr);


--| Procedure Name:  SET_RULE_FIRED
--| Inputs: The current rule just fired in the ES shell.
--| Outputs: None
--| Purpose: Sets appropriate flags in trace of this rule to
--|   indicate the rule has been fired.
--| Notes: Called by Procedure FIRE_RULES in Package DEMONS of the
--|   ES shell.

  procedure Set_Rule_Fired (The_Rule : in Rules.Rule_Ptr);


--| Procedure Name:  SET_RULE_FAILED
--| Inputs: The current rule that just failed to fire in ES shell.
--| Outputs: None
--| Purpose: Sets appropriate flags in trace of this rule to
--|   indicate the rule has failed (didn't fire);
--| Notes: Called by Procedure FIRE_RULES in Package DEMONS of the
--|   ES shell.
```

```
  procedure Set_Rule_Failed (The_Rule : in Rules.Rule_Ptr);

end Ef;  --spec
```

```
--|*****************************************************************
--|*****************************************************************
--|*****************************************************************
--|**********                                             **********
--|**********        EXPLANATION  FACILITY  BODY          **********
--|**********                                             **********
--|*****************************************************************
--|*****************************************************************
--|*****************************************************************

with Stack_Sequential_Bounded_Managed_Noniterative,
     Text_Io,
     Frames,
     Rules,
     User_Io;

package body Ef is

--|*****************************************************************
--|**********                                             **********
--|**********              TYPE DEFINITIONS               **********
--|**********                                             **********
--|*****************************************************************

--| Type MENU_OPTIONS_TYPE is an array of strings containing menu
--| options used in displaying a particular menu to the screen.

  type Menu_Options_Type is
            array(Positive range <>) of Frames.New_String;

--| Type EXEC_TRACE_TYPE is the record of information kept for
--| each rule processed by the ES shell.

  type Exec_Trace_Type;
  type Exec_Trace_Ptr is Access Exec_Trace_Type;

  type Exec_Trace_Type is
    record
      Curr_Frame : Frames.Frame_Ptr := null;
      Curr_Rule  : Rules.Rule_Ptr := null;
      Pending    : Boolean := True;
      Fired      : Boolean := False;
    end record;


--|*****************************************************************
--|**********                                             **********
--|**********          PACKAGE INSTANTIATIONS             **********
--|**********                                             **********
--|*****************************************************************
```

```
--| Miscellaneous instantiations of packages needed for screen I/O.

  package Pos_Io is new
          Text_Io.Integer_Io(Positive);
  package Rule_Io is new
          Text_Io.Integer_Io(Frames.Rule_No_Type);
  package Bool_Io is new
          Text_Io.Enumeration_Io(Boolean);
  package Rule_Op_Io is new
          Text_Io.Enumeration_Io(Rules.Rule_Operator);


--| The instantiation of the generic Stack package, creating the
--| EXECUTION TRACE.

  package The_Execution_Trace is new
    Stack_Sequential_Bounded_Managed_Noniterative(Exec_Trace_Ptr);


--|*****************************************************************
--|**********                                             **********
--|**********          VARIABLE DECLARATIONS              **********
--|**********                                             **********
--|*****************************************************************

--| Declaration of the size of the execution trace stack.

  The_Size : Positive := 300;


--| Declaration of the execution trace to be used by the EF.

  Exec_Trace : The_Execution_Trace.Stack(The_Size);


--|*****************************************************************
--|**********                                             **********
--|**********                EXCEPTIONS                   **********
--|**********                                             **********
--|*****************************************************************

--| An exception that is raised when the size of the execution trace
--| stack is not large enough.

  Ef_Fatal_Error : exception;


--|*****************************************************************
--|**********                                             **********
--|**********          ESTABLISH VISIBILITY               **********
--|**********                                             **********
--|*****************************************************************
```

```
--| Establish the visibility of different FRAME and RULE structures.

  function "=" (X, Y : in Rules.Rule_Ptr) return Boolean
    renames Rules."=";
  function "=" (X, Y : in Frames.Slot_Ptr) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Frames.Frame_Ptr) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Rules.Rule_Operand) return Boolean
    renames Rules."=";
  function "=" (X, Y : in Rules.If_Token_Ptr) return Boolean
    renames Rules."=";
  function "=" (X, Y : in Rules.Then_Token_Ptr) return Boolean
    renames Rules."=";
  function "=" (X, Y : in Frames.Frame_List_Ptr) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Frames.Frame_Name) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Frames.Frame_Type) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Frames.Slot_Name) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Frames.Parent_Ptr) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Frames.Demon_Ptr) return Boolean
    renames Frames."=";
  function "=" (X, Y : in Frames.Rule_List_Ptr) return Boolean
    renames Frames."=";


--|*****************************************************************
--|**********                                             **********
--|**********             UTILITY ROUTINES                **********
--|**********                                             **********
--|*****************************************************************

--| Subroutine Name:  WAIT
--| Inputs: None
--| Outputs: None
--| Purpose: Provide control of screen output.  Allows user to view
--|   information presented to the screen without it scrolling by.
--| Notes: None

  procedure Wait is
    X : String(1 .. 2);
    Y : Natural;
  begin
    Text_Io.New_Line;
    Text_Io.Put("     (Press <return> to continue)");
    Text_Io.Get_Line(X,Y);
    Text_Io.Skip_Line;
```

```
    Text_Io.New_Line(2);
  end Wait;


--| Subroutine Name:  FIND_TRACE_DATA
--| Inputs: The current rule being processed by the ES shell.
--| Outputs: Pointer to trace record of current rule.
--| Purpose: Search execution trace for current rule so that
--|   appropriate flags can be updated.
--| Notes: None

  function Find_Trace_Data (The_Rule : in Rules.Rule_Ptr)
                            return Exec_Trace_Ptr is
    Temp_Data : Exec_Trace_Ptr;
    Temp_Trace : The_Execution_Trace.Stack(The_Size);
  begin
    The_Execution_Trace.Copy(Exec_Trace, Temp_Trace);
    while not The_Execution_Trace.Is_Empty(Temp_Trace) loop
      Temp_Data := The_Execution_Trace.Top_Of(Temp_Trace);
      if Temp_Data.Curr_Rule.Rule_No = The_Rule.Rule_No then
        return Temp_Data;
      end if;
      The_Execution_Trace.Pop(Temp_Trace);
    end loop;
  end Find_Trace_Data;


--| Subroutine Name:  PRINT_A_RULE
--| Inputs: The rule whose contents are to be printed.
--| Outputs: The contents of the input rule.
--| Purpose: Print the contents of a rule to the screen in user
--|   readable format.
--| Notes: None

  procedure Print_A_Rule (The_Rule : in Rules.Rule_Ptr) is
    Temp_Rule : Rules.Rule_Ptr := The_Rule;
    Temp_Ante : Rules.If_Token_Ptr;
  begin
    if Temp_Rule.Antecedent = null then
      Temp_Ante := null;
    else
      Temp_Ante := new Rules.If_Token;
      Temp_Ante.all := Temp_Rule.Antecedent.all;
    end if;
    User_Io.Print_Rule_No(Temp_Rule.Rule_No);
    Text_Io.New_Line;
    Text_Io.Put("If");
    if Temp_Ante = null then
      Text_Io.Set_Col(5);
      Text_Io.Put("TRUE  Then");
      Text_Io.Set_Col(30);
      Text_Io.Put("-- since this rule has no antecedent");
```

```
      Text_Io.New_Line;
    else
      while Temp_Ante /= null loop
        Text_Io.Set_Col(5);
        User_Io.Print_Operand(Temp_Ante.Operand_1);
        Text_Io.Put("  ");
        Rule_Op_Io.Put(Temp_Ante.Operator);
        Text_Io.Put("  ");
        User_Io.Print_Operand(Temp_Ante.Operand_2);
        if Temp_Ante.Next = null then
          Text_Io.Put("   Then");
        else
          Text_Io.Put("   And");
        end if;
        Text_Io.New_Line;
        Temp_Ante := Temp_Ante.Next;
      end loop;
    end if;
    Text_Io.New_Line;
    User_Io.Print_Consequent(Temp_Rule.Consequent);
  end Print_A_Rule;


--| Subroutine Name:  FIND_SUB_FRAMES
--| Inputs: The name of the frame whose sibling frames are wanted.
--| Outputs: A list of pointers to all frame(s) subordinate to input
--|   frame. (possibly empty)
--| Purpose: Search active frames list for all frames having input
--|   frame name identified as one of their parents.
--| Notes: None

  function Find_Sub_Frames (The_Name : in Frames.Frame_Name)
                           return Frames.Frame_List_Ptr is
    The_Frames : Frames.Frame_List_Ptr;
    Head, Prev, Curr : Frames.Frame_List_Ptr;
    Is_Head : Boolean := True;
    Temp_Parents : Frames.Parent_Ptr;
  begin
    The_Frames := Frames.Active_Frames;
    while The_Frames /= null loop
      Temp_Parents := The_Frames.This_Frame.Parents;
      while Temp_Parents /= null loop
        if Temp_Parents.Parent_Name = The_Name then
          if Is_Head then
            Head := new Frames.Frame_List;
            Head.This_Frame := The_Frames.This_Frame;
            Prev := Head;
            Is_Head := False;
          else
            Curr := new Frames.Frame_List;
            Curr.This_Frame := The_Frames.This_Frame;
            Prev.Next := Curr;
```

```
            Prev := Curr;
          end if;
        end if;
        Temp_Parents := Temp_Parents.Next;
      end loop;
      The_Frames := The_Frames.Next;
    end loop;
    return Head;
  end Find_Sub_Frames;


--| Subroutine Name:  PRINT_PARENT_OF_INFO
--| Inputs: A pointer to the "parent" frame.
--| Outputs: The names of any/all sibling frames.
--| Purpose: Identify the frames for which the input frame is
--|   a parent. If there are none, output appropriate message ...
--|   otherwise print the sibling frame name(s).
--| Notes: None

  procedure Print_Parent_Of_Info
            (The_Object : in Frames.Frame_Ptr) is
    Temp_Object : Frames.Frame_Ptr := The_Object;
    Parent_Of_List : Frames.Frame_List_Ptr;
  begin
    Parent_Of_List := Find_Sub_Frames(Temp_Object.Name);
    if Parent_Of_List = null then
      User_Io.Print_Frame_Name(Temp_Object.Name);
      Text_Io.Put(" has no sub-class object(s) related to it.");
      Text_Io.New_Line;
      Text_Io.Put("     ( ");
      User_Io.Print_Frame_Name(Temp_Object.Name);
      Text_Io.Put(" is not a parent of any other object(s) )");
      Text_Io.New_Line;
    else
      User_Io.Print_Frame_Name(Temp_Object.Name);
      Text_Io.Put(" is the parent of ");
      if Parent_Of_List.Next = null then
        User_Io.Print_Frame_Name(Parent_Of_List.This_Frame.Name);
      else
        while Parent_Of_List /= null loop
          Text_Io.New_Line;
          Text_Io.Set_Col(5);
          User_Io.Print_Frame_Name(Parent_Of_List.This_Frame.Name);
          Parent_Of_List := Parent_Of_List.Next;
        end loop;
      end if;
      Text_Io.New_Line;
    end if;
  end Print_Parent_Of_Info;


--| Subroutine Name:  PRINT_MEMBER_OF_INFO
```

```
--| Inputs: A pointer to the frame in question.
--| Outputs: The frame names of any/all "parents" of input frame.
--| Purpose: Identify the parent frames of the input frame and print
--|   them to screen.  If there are none, output appropriate message
--|   ... otherwise print the frame name(s) of the parents.
--| Notes: None

  procedure Print_Member_Of_Info
            (The_Object : in Frames.Frame_Ptr) is
    Temp_Object : Frames.Frame_Ptr := The_Object;
  begin
    if Temp_Object.Parents = null then
      User_Io.Print_Frame_Name(Temp_Object.Name);
      Text_Io.Put
        (" is not a sub-class object of a higher class object.");
      Text_Io.New_Line;
      Text_Io.Put("     ( ");
      User_Io.Print_Frame_Name(Temp_Object.Name);
      Text_Io.Put(" has no parent(s) )");
      Text_Io.New_Line;
    else
      User_Io.Print_Frame_Name(Temp_Object.Name);
      Text_Io.Put(" is a sub_class object of ");
      if Temp_Object.Parents.Next = null then
        User_Io.Print_Frame_Name(Temp_Object.Parents.Parent_Name);
      else
        while Temp_Object.Parents /= null loop
          Text_Io.New_Line;
          Text_Io.Set_Col(5);
          User_Io.Print_Frame_Name(Temp_Object.Parents.Parent_Name);
          Temp_Object.Parents := Temp_Object.Parents.Next;
        end loop;
      end if;
      Text_Io.New_Line;
    end if;
  end Print_Member_Of_Info;


--| Subroutine Name:  FIND_OBJECT
--| Inputs: The name of a frame.
--| Outputs: Pointer to frame record of input frame name.
--| Purpose: Search the active frames list for the frame name input
--|   and return its record of information if found.
--| Notes: None

  function Find_Object (The_Name : in Frames.Frame_Name)
                        return Frames.Frame_Ptr is
    The_Frames : Frames.Frame_List_Ptr;
  begin
    The_Frames:= Frames.Active_Frames;
    while The_Frames /= null loop
      if The_Frames.This_Frame.Name = The_Name then
```

```
        return The_Frames.This_Frame;
      else
        The_Frames := The_Frames.Next;
      end if;
    end loop;
    return null;
  end Find_Object;


--| Subroutine Name:  FIND_ATTRIBUTE
--| Inputs: The name of an attribute (slot).
--| Outputs: Pointer to frame containing the input attribute name.
--| Purpose: Find the frame containing the attribute identified.
--| Notes: Searches all slot pointers of each frame in active list
--|   looking for a match to the input name.

  function Find_Attribute (The_Name : in Frames.Slot_Name)
                           return Frames.Frame_Ptr is
    The_Frames : Frames.Frame_List_Ptr;
    The_Slot : Frames.Slot_Ptr;
  begin
    The_Frames:= Frames.Active_Frames;
    while The_Frames /= null loop
      The_Slot := Frames.Find_Slot(The_Frames.This_Frame, The_Name);
      if The_Slot /= null then
        return The_Frames.This_Frame;
      else
        The_Frames := The_Frames.Next;
      end if;
    end loop;
    return null;
  end Find_Attribute;


--| Subroutine Name:  FIND_RULE_ATTRIBUTE
--| Inputs: An execution trace record pointer.
--| Outputs: Pointer to a slot record.
--| Purpose: Given a rule number (contained in the execution trace
--|   record), find the slot related to this rule number.
--| Notes: None

  function Find_Rule_Attribute (The_Data : in Exec_Trace_Ptr)
                                return Frames.Slot_Ptr is
    Temp_Data : Exec_Trace_Ptr := The_Data;
    Temp_Slot : Frames.Slot_Ptr;
    Temp_Demon : Frames.Demon_Ptr;
    Temp_Rules : Frames.Rule_List_Ptr;
  begin
    Temp_Slot := Temp_Data.Curr_Frame.Slots;
    while Temp_Slot /= null loop
      Temp_Demon := Temp_Slot.Demons;
      while Temp_Demon /= null loop
```

```
        Temp_Rules := Temp_Demon.Rules;
        while Temp_Rules /= null loop
          if Temp_Rules.Rule_No = Temp_Data.Curr_Rule.Rule_No then
            return Temp_Slot;
          end if;
          Temp_Rules := Temp_Rules.Next;
        end loop;
        Temp_Demon := Temp_Demon.Next;
      end loop;
      Temp_Slot := Temp_Slot.Next;
    end loop;
  end Find_Rule_Attribute;


--| Subroutine Name:  PRINT_HOW_HEADER
--| Inputs: The rule number currently being explained.
--| Outputs: HOW header information.
--| Purpose: Output header information for the HOW explanation
--|   function.
--| Notes: None

  procedure Print_How_Header
            (The_Rule_Number : in Frames.Rule_No_Type) is
    The_Num : Frames.Rule_No_Type := The_Rule_Number;
  begin
    Text_Io.Set_Col(7);
    Text_Io.Put
    ("Currently Explaining HOW The System Decided To");
    Text_Io.Put
    (" Process Rule #");
    Rule_Io.Put(The_Num, 3);
    Text_Io.New_Line(2);
  end Print_How_Header;


--| Subroutine Name:  PRINT_GOAL_RULE_INFO
--| Inputs: An execution trace record.
--| Outputs: Explanation of HOW the system processed this rule.
--| Purpose: Explain HOW the system decided to process this rule
--|   (contained in the execution trace record) if this rule is
--|   associated with a GOAL type frame.
--| Notes: None

  procedure Print_Goal_Rule_Info (The_Data : in Exec_Trace_Ptr) is
    Temp_Data : Exec_Trace_Ptr := The_Data;
  begin
    Print_How_Header(Temp_Data.Curr_Rule.Rule_No);
    Text_Io.Put("Rule #");
    Rule_Io.Put(Temp_Data.Curr_Rule.Rule_No, 3);
    Text_Io.Put(" is associated with the Goal Frame = ");
    User_Io.Print_Frame_Name(Temp_Data.Curr_Frame.Name);
    Text_Io.New_Line;
```

```
    Text_Io.Put
      ("Attempting to determine a Goal Value for Attribute = ");
    User_Io.Print_Operand(Temp_Data.Curr_Rule.Consequent.Target);
    Text_Io.New_Line;
    Text_Io.Put("This value is to be obtained by ");
    case Temp_Data.Curr_Rule.Consequent.Source.Operand_Kind is
      when Rules.Local_Slot => Text_Io.Put("Inheritance from ");
      when Rules.Dist_Slot  => Text_Io.Put("Message to ");
      when others           => null;
    end case;
    Text_Io.Put("Attribute = ");
    User_Io.Print_Operand(Temp_Data.Curr_Rule.Consequent.Source);
    Text_Io.New_Line;
  end Print_Goal_Rule_Info;


--| Subroutine Name:  PRINT_START_RULE_INFO
--| Inputs: An execution trace record.
--| Outputs: Explanation of HOW the system processed this rule.
--| Purpose: Explain HOW the system decided to process this rule
--|   (contained in the execution trace record) if this rule is
--|   associated with a START type frame.
--| Notes: None

  procedure Print_Start_Rule_Info (The_Data : in Exec_Trace_Ptr) is
    Temp_Data : Exec_Trace_Ptr := The_Data;
  begin
    Print_How_Header(Temp_Data.Curr_Rule.Rule_No);
    Text_Io.Put("Rule #");
    Rule_Io.Put(Temp_Data.Curr_Rule.Rule_No, 3);
    Text_Io.Put(" is associated with the Start Frame = ");
    User_Io.Print_Frame_Name(Temp_Data.Curr_Frame.Name);
    Text_Io.New_Line;
    Text_Io.Put_Line
    ("Using initial information this frame provides, begin");
    Text_Io.Put("processing by attempting to determine a ");
    Text_Io.Put("Value for ");
    User_Io.Print_Operand(Temp_Data.Curr_Rule.Consequent.Target);
    Text_Io.New_Line;
    Text_Io.Put("This value is to be obtained by ");
    case Temp_Data.Curr_Rule.Consequent.Source.Operand_Kind is
      when Rules.Local_Slot => Text_Io.Put("Inheritance from ");
      when Rules.Dist_Slot  => Text_Io.Put("Message to ");
      when others           => null;
    end case;
    Text_Io.Put("attribute ");
    User_Io.Print_Operand(Temp_Data.Curr_Rule.Consequent.Source);
    Text_Io.New_Line;
  end Print_Start_Rule_Info;


--| Subroutine Name:  PRINT_PROMPT_RULE_INFO
```

```
--| Inputs: An execution trace record.
--| Outputs: Explanation of HOW the system processed this rule.
--| Purpose: Explain HOW the system decided to process this rule
--|   (contained in the execution trace record) if this rule is a
--|   PROMPT type rule.
--| Notes: None

  procedure Print_Prompt_Rule_Info (The_Data : in Exec_Trace_Ptr;
                                    The_Trace : The_Execution_Trace.Stack) is
    Curr_Data : Exec_Trace_Ptr := The_Data;
    Pending_Data : Exec_Trace_Ptr;
    Temp_Trace : The_Execution_Trace.Stack(The_Size);
  begin
    The_Execution_Trace.Copy(The_Trace, Temp_Trace);
    while not The_Execution_Trace.Is_Empty(Temp_Trace) loop
      Pending_Data := The_Execution_Trace.Top_Of(Temp_Trace);
      if Pending_Data.Pending then
        exit;
      end if;
      The_Execution_Trace.Pop(Temp_Trace);
    end loop;
    Text_Io.Put("Rule #");
    Rule_Io.Put(Pending_Data.Curr_Rule.Rule_No, 3);
    Text_Io.Put(" is related to Rule #");
    Rule_Io.Put(Curr_Data.Curr_Rule.Rule_No, 3);
    Text_Io.New_Line(2);
    Text_Io.Put("Rule #");
    Rule_Io.Put(Pending_Data.Curr_Rule.Rule_No, 3);
    Text_Io.Put(" is in 'pending' status awaiting a value for ");
    User_Io.Print_Operand(Curr_Data.Curr_Rule.Consequent.Target);
    Text_Io.New_Line(2);
    Print_A_Rule(Pending_Data.Curr_Rule);
    Text_Io.New_Line(2);
    Text_Io.Put("Remember ... Rule #");
    Rule_Io.Put(Curr_Data.Curr_Rule.Rule_No, 3);
    Text_Io.Put(" prompts for a value for ");
    User_Io.Print_Operand(Curr_Data.Curr_Rule.Consequent.Target);
    Text_Io.New_Line(2);
  end Print_Prompt_Rule_Info;


--| Subroutine Name:  PRINT_STANDARD_RULE_INFO
--| Inputs: An execution trace record.
--| Outputs: Explanation of HOW the system processed this rule.
--| Purpose: Explain HOW the system decided to process this rule
--|   (contained in the execution trace record) if this rule is a
--|   LOCAL_SLOT or DIST_SLOT type rule.
--| Notes: None

  procedure Print_Standard_Rule_Info
            (The_Data : in Exec_Trace_Ptr) is
    Temp_Data : Exec_Trace_Ptr := The_Data;
```

```
    Curr_Attrib : Frames.Slot_Ptr;
  begin
    Curr_Attrib := Find_Rule_Attribute(Temp_Data);
    Print_How_Header(Temp_Data.Curr_Rule.Rule_No);
    Text_Io.Put("Attempting to determine value for Attribute = ");
    User_Io.Print_Slot_Name(Curr_Attrib.Name);
    Text_Io.New_Line;
    Text_Io.Put("Rule #");
    Rule_Io.Put(Temp_Data.Curr_Rule.Rule_No, 3);
    Text_Io.Put(" is one of the rules that attempts to do this");
    Text_Io.New_Line(2);
    Print_A_Rule(Temp_Data.Curr_Rule);
    Text_Io.New_Line;
  end Print_Standard_Rule_Info;


--| Subroutine Name:  EXPLAIN_TOP_OF_TRACE
--| Inputs: The top record on the execution trace stack.
--|         The remainder of the execution trace stack.
--| Outputs: None
--| Purpose: Determine what type of rule is on the top of the
--|   execution trace and call the appropriate procedure to explain
--|   HOW this type of rule was processed.
--| Notes: None

  procedure  Explain_Top_Of_Trace (The_Top : in Exec_Trace_Ptr;
                   The_Trace : in out The_Execution_Trace.Stack) is
    Temp_Top : Exec_Trace_Ptr := The_Top;
  begin
    Text_Io.New_Line(5);
    if Temp_Top.Curr_Frame.F_Type = Frames.Goal then
      Print_Goal_Rule_Info(Temp_Top);
    elsif Temp_Top.Curr_Frame.F_Type = Frames.Start then
      Print_Start_Rule_Info(Temp_Top);
    elsif Temp_Top.Curr_Rule.Antecedent = null and then
          Temp_Top.Curr_Rule.Consequent.Source.Operand_Kind =
          Rules.Prompt then
      Print_How_Header(Temp_Top.Curr_Rule.Rule_No);
      Print_Prompt_Rule_Info(Temp_Top, The_Trace);
    elsif Temp_Top.Pending then
      Print_Standard_Rule_Info(Temp_Top);
    end if;
  end Explain_Top_Of_Trace;



--| Subroutine Name:  PRINT_STACK_TO_SMALL
--| Inputs: None
--| Outputs: An error message.
--| Purpose: Print an appropriate error message if the execution
--|   trace stack is not large enough to hold all of the necessary
--|   trace records.
```

```
--| Notes: None

  procedure Print_Stack_To_Small is
  begin
    Text_Io.New_Line(30);
    Text_Io.Set_Col(27);
    Text_Io.Put_Line("*****   FATAL  ERROR  *****");
    Text_Io.Set_Col(20);
    Text_Io.Put_Line("Stack size for Execution Trace to small.");
    Text_Io.Set_Col(30);
    Text_Io.Put("Current size =");
    Pos_Io.Put(The_Size, 5);
    Text_Io.New_Line;
    Text_Io.Set_Col(14);
    Text_Io.Put_Line
    ("To correct, increase THE_SIZE in package body of EF ");
    Text_Io.Set_Col(38);
    Text_Io.Put_Line("and");
    Text_Io.Set_Col(29);
    Text_Io.Put_Line("Recompile the system.");
    Text_Io.New_Line(6);
    raise Ef_Fatal_Error;
  end Print_Stack_To_Small;


--|*****************************************************************
--|**********                                             **********
--|**********              MENU  ROUTINES                 **********
--|**********                                             **********
--|*****************************************************************

--| Subroutine Name:  FIND_LONGEST_OPTION
--| Inputs: An array of menu option strings.
--| Outputs: The length of the longest string in the array.
--| Purpose: Determine length of longest option for a particular
--|   menu.
--| Notes: Used to center the menu on the screen, when it is
--|   displayed.

  function Find_Longest_Option
           (The_Array : in Menu_Options_Type) return Positive is
    Longest : Positive := 1;
  begin
    for I in The_Array'range loop
      if The_Array(I).The_Length = 0 then
        exit;
      elsif The_Array(I).The_Length > Longest then
        Longest := The_Array(I).The_Length;
      end if;
    end loop;
    return Longest;
  end Find_Longest_Option;
```

```
--| Subroutine Name:  DISPLAY_A_MENU
--| Inputs: An array of menu option strings.
--|         The total number of strings input.
--| Outputs: Numeric value of menu option selected by the user.
--| Purpose: Print a menu to the screen and get the desired option
--|   selected by the user.
--| Notes: The menu is centered on the screen (assumes 80 char wide
--|   screen). Error checking on user selection performed.  Illegal
--|   values aren't accepted.  User is re-prompted for another
--|   selection if previous selection was invalid.

  function Display_A_Menu
           (The_Options_List : in Menu_Options_Type;
            Max_Options : in Positive) return Positive is
    Title_Offset : Integer := 0;
    Options_Offset : Integer := 0;
    Longest_Option : Positive :=
                     Find_Longest_Option(The_Options_List);
    Choice : Positive := 1;
  begin
    Text_Io.New_Line(2);
    Title_Offset := (80 - The_Options_List(1).The_Length)/2;
    Options_Offset := (80 - Longest_Option)/2;
    Text_Io.Set_Col(Text_Io.Positive_Count(Title_Offset));
    Text_Io.Put(The_Options_List(1).The_String
                              (1 .. The_Options_List(1).The_Length));
    Text_Io.New_Line(2);
    for I in 2 .. Max_Options + 1 loop
      Text_Io.Set_Col(Text_Io.Positive_Count(Options_Offset));
      Text_Io.Put(The_Options_List(I).The_String
                              (1 .. The_Options_List(I).The_Length));
      Text_Io.New_Line;
    end loop;
    Text_Io.New_Line;
    loop
      begin
        Text_Io.Set_Col(Text_Io.Positive_Count(Options_Offset));
        Text_Io.Put(" enter desired option:");
        Pos_Io.Get(Choice);
        if Choice > Max_Options then
          Text_Io.Put_Line("Invalid Option!  Please Try Again.");
          Text_Io.Skip_Line;
          Text_Io.New_Line;
        else
          exit;
        end if;
      exception
        when Text_Io.Data_Error =>
           Text_Io.New_Line;
           Text_Io.Put_Line("Invalid Option!  Please Try Again.");
```

```
            Text_Io.Skip_Line;
            Text_Io.New_Line;
        end;
      end loop;
      return Choice;
    end Display_A_Menu;


--| Subroutine Name:  SETUP_CONTINUE_HOW_MENU
--| Inputs: None
--| Outputs: Array of options for the Continue Explaining How menu.
--| Purpose: Input the menu options text for the Continue Explaining
--|   How menu.
--| Notes: The HOW explanation can be repeated as often as the user
--|   needs.  These menu options provide the user this capability.

    function Setup_Continue_How_Menu return Menu_Options_Type is
      Opts : Menu_Options_Type(1 .. 5);
    begin
      Opts(1).The_String(1 .. 35) :=
              "***** Continue Explaining HOW *****";
      Opts(1).The_Length := 35;
      Opts(2).The_String(1 .. 36) :=
              "1)  CONTINUE How Explanation Process";
      Opts(2).The_Length := 36;
      Opts(3).The_String(1 .. 32) :=
              "2)  EXIT How Explanation Process";
      Opts(3).The_Length := 32;
      return Opts;
    end Setup_Continue_How_Menu;


--| Subroutine Name:  SETUP_ACCESS_TO_EF_MENU
--| Inputs: None
--| Outputs: Array of options for the Access EF menu.
--| Purpose: Input the menu options text for the Access EF menu.
--| Notes: These options provide the user with runtime access to and
--|   from the explanation facility.

    function Setup_Access_To_Ef_Menu return Menu_Options_Type is
      Opts : Menu_Options_Type(1 .. 5);
    begin
      Opts(1).The_String(1 .. 26) := "*****  USER OPTIONS  *****";
      Opts(1).The_Length := 26;
      Opts(2).The_String(1 .. 23) := "1)  Request Explanation";
      Opts(2).The_Length := 23;
      Opts(3).The_String(1 .. 30) := "2)  Provide Response To Prompt";
      Opts(3).The_Length := 30;
      return Opts;
    end Setup_Access_To_Ef_Menu;
```

```
--| Subroutine Name:  SETUP_MAIN_OPTIONS_MENU
--| Inputs: None
--| Outputs: Array of options for the Main Explanation menu.
--| Purpose: Input the menu options text for the Main Explanation
--|   menu.
--| Notes: These options identify the main runtime explanation
--|   functions available to the user.

  function Setup_Main_Options_Menu return Menu_Options_Type is
    Opts : Menu_Options_Type(1 .. 7);
  begin
    Opts(1).The_String(1 .. 41) :=
            "*****  RUNTIME EXPLANATION OPTIONS  *****";
    Opts(1).The_Length := 41;
    Opts(2).The_String(1 .. 56) :=
      "1)  SHOW RULE (show the contents of an individual rule).";
    Opts(2).The_Length := 56;
    Opts(3).The_String(1 .. 48) :=
      "2)  WHY (does the system want this information)?";
    Opts(3).The_Length := 48;
    Opts(4).The_String(1 .. 64) :=
      "3)  HOW (did system get to this point in reasoning process)?";
    Opts(4).The_Length := 64;
    Opts(5).The_String(1 .. 46) :=
      "4)  WHAT (is the value of attribute (slot) X)?";
    Opts(5).The_Length := 46;
    Opts(6).The_String(1 .. 59) :=
      "5)  WHERE (is object X or attribute X found in the system)?";
    Opts(6).The_Length := 59;
    Opts(7).The_String(1 .. 35) :=
      "6)  EXIT (the explanation facility)";
    Opts(7).The_Length := 35;
    return Opts;
  end Setup_Main_Options_Menu;


--| Subroutine Name:  SETUP_EOP_OPTIONS_MENU
--| Inputs: None
--| Outputs: Array of options for the End-Of-Processing menu.
--| Purpose: Input the menu options text for the End-Of-Processing
--|   menu.
--| Notes: These options identify the explanation functions available
--|   to the user once the ES shell has completed execution.

  function Setup_Eop_Options_Menu return Menu_Options_Type is
    Opts : Menu_Options_Type(1 .. 7);
  begin
    Opts(1).The_String(1 .. 51) :=
      "*****  END-OF-PROCESSING EXPLANATION OPTIONS  *****";
    Opts(1).The_Length := 51;
    Opts(2).The_String(1 .. 56) :=
      "1)  SHOW RULE (show the contents of an individual rule).";
```

```
    Opts(2).The_Length := 56;
    Opts(3).The_String(1 .. 46) :=
      "2)  WHAT (is the value of attribute (slot) X)?";
    Opts(3).The_Length := 46;
    Opts(4).The_String(1 .. 59) :=
      "3)  WHERE (is object X or attribute X found in the system)?";
    Opts(4).The_Length := 59;
    Opts(5).The_String(1 .. 48) :=
      "4)  SHOW (critical path (all rules that fired) )";
    Opts(5).The_Length := 48;
    Opts(6).The_String(1 .. 46) :=
      "5)  SHOW (execution trace (all rules tested) )";
    Opts(6).The_Length := 46;
    Opts(7).The_String(1 .. 35) :=
      "6)  EXIT (the explanation facility)";
    Opts(7).The_Length := 35;
    return Opts;
  end Setup_Eop_Options_Menu;


--|*****************************************************************
--|**********                                             **********
--|**********         MAIN  PROCESSING  ROUTINES          **********
--|**********                                             **********
--|*****************************************************************


--| Subroutine Name:  DISPLAY_A_RULE
--| Inputs: None
--| Outputs: User prompt for a rule number.
--| Purpose: Print the contents of a specified rule to the screen.
--| Notes: Prompts the user for a rule number.  Searches the ES
--|   shell rule list for this rule.  Calls Print_A_Rule to output
--|   the rule's contents if found.  Prints error message if rule
--|   not found. Also checks for invalid data entry, allowing only
--|   positive numbers to to be input.  Re-prompts user if invalid
--|   input is received.

  procedure Display_A_Rule is
    The_Rule : Rules.Rule_Ptr;
    Rule_Num : Frames.Rule_No_Type;
  begin
    loop
      begin
        Text_Io.New_Line;
        Text_Io.Put("enter rule number to display:");
        Rule_Io.Get(Rule_Num);
        The_Rule := Rules.Find_Rule(Rule_Num);
        if The_Rule = null then
          Text_Io.New_Line;
          Text_Io.Put_Line("ERROR::Invalid Rule Number!");
          Text_Io.Put("     Rule #");
```

```
            Rule_Io.Put(Rule_Num,3);
            Text_Io.Put(" Not Found In Active Rule List!");
            Text_Io.New_Line(2);
          else
            Text_Io.New_Line(30);
            Print_A_Rule(The_Rule);
            exit;
          end if;
          exception
            when Constraint_Error | Numeric_Error | Text_Io.Data_Error
                 => Text_Io.Put_Line
                ("Invalid entry ... enter a positive number only.");
              Text_Io.Skip_Line;
              Text_Io.New_Line(2);
      end;
    end loop;
  end Display_A_Rule;


--| Subroutine Name:  DUMP_EXECUTION_TRACE
--| Inputs: The execution trace to be dumped.
--| Outputs: The contents of the execution trace.
--| Purpose: Output the contents of the execution trace.
--| Notes: Loops throught the execution trace (until it is empty)
--|   and prints the rule number of each trace record and indicates
--|   whether or not the rule "is pending", "fired" or "failed".

  procedure Dump_Execution_Trace (The_Trace : in
                                  The_Execution_Trace.Stack) is
    T_Trace : The_Execution_Trace.Stack(The_Size);
    Temp : Exec_Trace_Ptr;
    Count : Natural := 0;
  begin
    The_Execution_Trace.Copy(The_Trace, T_Trace);
    Text_Io.New_Line;
    while not The_Execution_Trace.Is_Empty(T_Trace) loop
      Temp := The_Execution_Trace.Top_Of(T_Trace);
      The_Execution_Trace.Pop(T_Trace);
      Text_Io.Put("Rule #");
      Rule_Io.Put(Temp.Curr_Rule.Rule_No, 3);
      Text_Io.Set_Col(15);
      if Temp.Pending then
        Text_Io.Put("Pending");
      elsif Temp.Fired then
        Text_Io.Put("Fired");
      else
        Text_Io.Put("Failed");
      end if;
      Text_Io.New_Line(2);
      Count := Count + 2;
      if Count >= 20 then
        Count := 0;
```

```
        Wait;
      end if;
    end loop;
  end Dump_Execution_Trace;


--| Subroutine Name:  DUMP_CRITICAL_PATH
--| Inputs: The execution trace.
--| Outputs: All rules that have "fired" flag set = true.
--| Purpose: Print the critical path the ES shell took to arrive at
--|   its decision.  (all rules that were tested and fired)
--| Notes: Loops through the execution trace (until if is empty) and
--|   outputs the rule numbers of all rules noted as having been
--|   fired by the ES shell.

  procedure Dump_Critical_Path(The_Trace : in
                               The_Execution_Trace.Stack) is
    T_Trace : The_Execution_Trace.Stack(The_Size);
    Temp : Exec_Trace_Ptr;
    Count : Natural := 0;
    First : Boolean := True;
  begin
    The_Execution_Trace.Copy(The_Trace, T_Trace);
    while not The_Execution_Trace.Is_Empty(T_Trace) loop
      Temp := The_Execution_Trace.Top_Of(T_Trace);
      if Temp.Fired then
        The_Execution_Trace.Pop(T_Trace);
        if First then
          Text_Io.Put("GOAL ==>");
          First := False;
        end if;
        if The_Execution_Trace.Is_Empty(T_Trace) then
          Text_Io.Put("START ==>");
        end if;
        Text_Io.Put("  Rule #");
        Rule_Io.Put(Temp.Curr_Rule.Rule_No, 3);
        Text_Io.New_Line(2);
        Count := Count + 2;
        if Count >= 22 then
          Count := 0;
          Wait;
        end if;
      else
        The_Execution_Trace.Pop(T_Trace);
      end if;
    end loop;
    Wait;
  end Dump_Critical_Path;


--| Subroutine Name:  EXPLAIN_WHAT
--| Inputs: None
```

```
--| Outputs: The value of an attribute (slot) in the knowledge base.
--| Purpose: Print the value of an attribute in the knowledge base.
--| Notes: Prompts the user for the name of an attribute.  Searches
--|   the knowledge base for this attribute name.  If found, prints
--|   the value of the attribute.  If not found, prints appropriate
--|   error message.

  procedure Explain_What is
    Found : Boolean := False;
    The_Slot_Name : Frames.Slot_Name;
    The_Frames : Frames.Frame_List_Ptr;
    The_Slot : Frames.Slot_Ptr;
  begin
    The_Frames:= Frames.Active_Frames;
    Text_Io.New_Line(5);
    Text_Io.Put("Enter NAME of ATTRIBUTE whose value is desired: ");
    The_Slot_Name := User_Io.Read_Slot_Name;
    Text_Io.Skip_Line;
    Text_Io.New_Line(2);
    while The_Frames /= null loop
      The_Slot := Frames.Find_Slot(The_Frames.This_Frame,
                                   The_Slot_Name);
      if The_Slot = null then
        The_Frames := The_Frames.Next;
      else
        Found := True;
        Text_Io.Set_Col(5);
        Text_Io.Put("The Current Value Of ");
        User_Io.Print_Slot_Name(The_Slot_Name);
        Text_Io.Put(" = ");
        User_Io.Print_Value(The_Slot.Value);
        exit;
      end if;
    end loop;
    if not Found then
      Text_Io.Set_Col(5);
      Text_Io.Put("The ATTRIBUTE = ");
      User_Io.Print_Slot_Name(The_Slot_Name);
      Text_Io.Put(" was not found in the Knowledge Base.");
    end if;
    Text_Io.New_Line(2);
    Wait;
  end Explain_What;


--| Subroutine Name:  EXPLAIN_WHERE
--| Inputs: None
--| Outputs: The relationship(s) of an attribute (slot) or object
--|   (frame) in the knowledge base.
--| Purpose: Print the relationship(s) of an attribute or object in
--|   the knowledge base.
--| Notes: Prompts the user for a name.  Assumes it is an object name
```

```
--|   initially.  Searches the knowledge base for an object with this
--|   name.  If found, outputs the parent and/or sibling object(s)
--|   related to this frame.  If not found, assumes name is an
--|   attribute name.  Searches for this attribute in the knowledge
--|   base and if found, identifies the object to which the attribute
--|   is related.

  procedure Explain_Where is
    The_Object_Name : Frames.Frame_Name;
    The_Attrib_Name : Frames.Slot_Name;
    Temp_Object : Frames.Frame_Ptr;
  begin
    Text_Io.New_Line(5);
    Text_Io.Put_Line
      ("Enter name of Object/Attribute whose relationships");
    Text_Io.Put("within the Knowledge Base are to be identified: ");
    The_Object_Name := User_Io.Read_Frame_Name;
    Text_Io.Skip_Line;
    Text_Io.New_Line(30);
    Temp_Object := Find_Object(The_Object_Name);
    if Temp_Object /= null then
      Text_Io.Set_Col(15);
      Text_Io.Put("Identifying the relationships of OBJECT = ");
      User_Io.Print_Frame_Name(The_Object_Name);
      Text_Io.New_Line(2);
      Print_Member_Of_Info(Temp_Object);
      Text_Io.New_Line;
      Print_Parent_Of_Info(Temp_Object);
      Text_Io.New_Line;
    else
      The_Attrib_Name := Frames.Slot_Name(The_Object_Name);
      Temp_Object := Find_Attribute(The_Attrib_Name);
      if Temp_Object = null then
        Text_Io.Put_Line("OOPS!  Something is wrong!");
        Text_Io.Put("The value input ... ");
        User_Io.Print_Slot_Name(The_Attrib_Name);
        Text_Io.Put(" ... was not found anywhere in the KB.");
        Text_Io.New_Line;
      else
        Text_Io.Set_Col(12);
        Text_Io.Put("Identifying the relationships of ATTRIBUTE = ");
        User_Io.Print_Slot_Name(The_Attrib_Name);
        Text_Io.New_Line(2);
        User_Io.Print_Slot_Name(The_Attrib_Name);
        Text_Io.Put(" is an attribute of the Object = ");
        User_Io.Print_Frame_Name(Temp_Object.Name);
        Text_Io.New_Line;
      end if;
    end if;
    Text_Io.New_Line;
    Wait;
  end Explain_Where;
```

```
--| Subroutine Name:  EXPLAIN_WHY
--| Inputs: None
--| Outputs: Explanation of WHY the information is being requested.
--| Purpose: Explain WHY the system is asking the user for the
--|   specific information.
--| Notes:

  procedure Explain_Why is
    Curr_Data : Exec_Trace_Ptr;
    Temp_Trace : The_Execution_Trace.Stack(The_Size);
  begin
    The_Execution_Trace.Copy(Exec_Trace, Temp_Trace);
    Curr_Data := The_Execution_Trace.Top_Of(Temp_Trace);
    The_Execution_Trace.Pop(Temp_Trace);
    Text_Io.New_Line(30);
    Text_Io.Put("Currently seeking value for ");
    User_Io.Print_Operand(Curr_Data.Curr_Rule.Consequent.Target);
    Text_Io.Put(" of Object = ");
    User_Io.Print_Frame_Name(Curr_Data.Curr_Frame.Name);
    Text_Io.New_Line;
    Text_Io.Put("Rule #");
    Rule_Io.Put(Curr_Data.Curr_Rule.Rule_No, 3);
    Text_Io.Put
      (" is the current rule prompting for this information.");
    Text_Io.New_Line;
    Text_Io.Put_Line("              HOWEVER");
    Print_Prompt_Rule_Info(Curr_Data, Temp_Trace);
    Wait;
  end Explain_Why;


--| Subroutine Name:  EXPLAIN_HOW
--| Inputs: The execution trace.
--| Outputs: Explanation of HOW the system decided to process the
--|   the particular rule in question (the top of the trace).
--| Purpose: Starting with the first rule tested, explain how this
--|   rule was processed.
--| Notes: A recursive routine that allows the user to determine how
--|   much of an explanation he/she desires.  After each explanation,
--|   the user has the choice of continuing with the HOW explanation
--|   or exiting this process.

  procedure Explain_How
            (The_Trace : in out The_Execution_Trace.Stack;
             Quit : in out Boolean) is
    Temp : Exec_Trace_Ptr;
    Choice : Positive;
    Temp_Trace : The_Execution_Trace.Stack(The_Size);
  begin
    The_Execution_Trace.Copy(The_Trace, Temp_Trace);
```

```
    Temp := The_Execution_Trace.Top_Of(The_Trace);
    The_Execution_Trace.Pop(The_Trace);
    if not The_Execution_Trace.Is_Empty(The_Trace) then
      Explain_How(The_Trace, Quit);
    end if;
    if not Quit then
      Explain_Top_Of_Trace(Temp, Temp_Trace);
      Wait;
      Text_Io.New_Line(2);
      Choice := Display_A_Menu(Setup_Continue_How_Menu, 2);
      case Choice is
        when 1 => null;
        when 2 => Quit := True;
        when others => null;
      end case;
    end if;
  end Explain_How;


--| Subroutine Name:  EXPLAIN_RUNTIME
--| Inputs: None
--| Outputs: None
--| Purpose: Select the appropriate runtime explanation function and
--|   execute it, based on the users selection from the runtime
--|   options menu.
--| Notes: None

  procedure Explain_Runtime is
    Temp_Trace : The_Execution_Trace.Stack(The_Size);
    Quit_How : Boolean := False;
    Option : Positive := Display_A_Menu
                        (The_Options_List => Setup_Main_Options_Menu,
                         Max_Options => 6);
  begin
    loop
      case Option is
        when 1 => Display_A_Rule;
                  Option := Display_A_Menu
                      (The_Options_List => Setup_Main_Options_Menu,
                       Max_Options => 6);
        when 2 => Explain_Why;
                  Option := Display_A_Menu
                      (The_Options_List => Setup_Main_Options_Menu,
                       Max_Options => 6);
        when 3 => The_Execution_Trace.Copy(Exec_Trace, Temp_Trace);
                  Explain_How(Temp_Trace, Quit_How);
                  Option := Display_A_Menu
                      (The_Options_List => Setup_Main_Options_Menu,
                       Max_Options => 6);
        when 4 => Explain_What;
                  Option := Display_A_Menu
                      (The_Options_List => Setup_Main_Options_Menu,
```

```
                    Max_Options => 6);
        when 5 => Explain_Where;
                  Option := Display_A_Menu
                      (The_Options_List => Setup_Main_Options_Menu,
                       Max_Options => 6);
        when 6 => exit;
        when others => null;
      end case;
    end loop;
  end Explain_Runtime;


--|*****************************************************************
--|**********                                             **********
--|**********          EXPORTED  ROUTINES                 **********
--|**********                                             **********
--|*****************************************************************



--| Subroutine Name:  BUILD_EXECUTION_TRACE
--| Inputs: The current frame being processed by the ES shell.
--|         The current rule being processed by the ES shell.
--| Outputs: None
--| Purpose: Build a new execution trace record, fill in the needed
--|   information and push it onto the execution trace stack.
--| Notes: Insures the execution trace stack is not to small.  Calls
--|   Print_Stack_To_Small to output instructive error msg if it is.

  procedure Build_Execution_Trace (The_Frame : in Frames.Frame_Ptr;
                                   The_Rule : in Rules.Rule_Ptr) is
    Temp_Data : Exec_Trace_Ptr;
  begin
    Temp_Data := new Exec_Trace_Type;
    Temp_Data.Curr_Frame := The_Frame;
    Temp_Data.Curr_Rule := The_Rule;
    The_Execution_Trace.Push(Temp_Data, Exec_Trace);
    exception
      when The_Execution_Trace.Overflow => Print_Stack_To_Small;
  end Build_Execution_Trace;


--| Subroutine Name:  SET_RULE_FIRED
--| Inputs: The current rule just fired in the ES shell.
--| Outputs: None
--| Purpose: Set the appropriate flags for the given rule to indicate
--|   that it has been fired ... its antecedent tested = true.
--| Notes: If the rule has no antecedent (therefore always true), the
--|   execution trace is searched until the trace record containing
--|   the rule is found and the appropriate indicators are then set.
--|   If the rule has an antecedent, then a new trace record is
--|   created, set equal to the trace record for the rule that is
```

```
--|   already on the stack, and this new record is also pushed
--|   onto the stack. This provides the capability of showing (in
--|   the execution trace) when a rule changes from a pending status
--|   to a fired status.

  procedure Set_Rule_Fired (The_Rule : in Rules.Rule_Ptr) is
    Temp_Rule : Rules.Rule_Ptr := The_Rule;
    Temp_Trace : Exec_Trace_Ptr;
    Temp_Data : Exec_Trace_Ptr;
  begin
    Temp_Trace := Find_Trace_Data(Temp_Rule);
    if Temp_Rule.Antecedent /= null then
      Temp_Data := new Exec_Trace_Type;
      Temp_Data.all := Temp_Trace.all;
      Temp_Data.Fired := True;
      Temp_Data.Pending := False;
      The_Execution_Trace.Push(Temp_Data, Exec_Trace);
    else
      Temp_Trace.Fired := True;
      Temp_Trace.Pending := False;
    end if;
    exception
      when The_Execution_Trace.Overflow => Print_Stack_To_Small;
  end Set_Rule_Fired;


--| Subroutine Name:  SET_RULE_FAILED
--| Inputs: The current rule that just failed in the ES shell.
--| Outputs: None
--| Purpose: Set the appropriate flags for the given rule to
--|   indicate that it has failed ... its antecedent tested = false.
--| Note: A new trace record is created, set equal to the trace
--|   record for the rule that is already on the stack, and this new
--|   record is also pushed onto the stack.
--|   This provides the capability of showing (in execution trace)
--|   when a rule changes from a pending status to a failed status.

  procedure Set_Rule_Failed (The_Rule : in Rules.Rule_Ptr) is
    Temp_Rule : Rules.Rule_Ptr := The_Rule;
    Temp_Trace : Exec_Trace_Ptr;
    Temp_Data : Exec_Trace_Ptr;
  begin
    Temp_Trace := Find_Trace_Data(Temp_Rule);
    Temp_Data := new Exec_Trace_Type;
    Temp_Data.all := Temp_Trace.all;
    Temp_Data.Pending := False;
    The_Execution_Trace.Push(Temp_Data, Exec_Trace);
    exception
      when The_Execution_Trace.Overflow => Print_Stack_To_Small;
  end Set_Rule_Failed;
```

```
--| Subroutine Name:  CLEAR_EXECUTION_TRACE
--| Inputs: None
--| Outputs: None
--| Purpose: Clear or re-initialize execution trace stack = empty.
--| Notes: None

  procedure Clear_Execution_Trace is
  begin
    The_Execution_Trace.Clear(Exec_Trace);
  end Clear_Execution_Trace;


--| Subroutine Name:  ACCESS_END_OF_PROCESSING
--| Inputs: None
--| Outputs: None
--| Purpose: Select the appropriate end-of-processing explanation
--|   function and execute it, based on the users selection from
--|   the end-of-processing options menu.
--| Notes: None

  procedure Access_End_Of_Processing_Ef is
    Option : Positive := Display_A_Menu
                      (The_Options_List => Setup_Eop_Options_Menu,
                       Max_Options => 6);
  begin
    loop
      case Option is
        when 1 => Display_A_Rule;
                  Option := Display_A_Menu
                       (The_Options_List => Setup_Eop_Options_Menu,
                        Max_Options => 6);
        when 2 => Explain_What;
                  Option := Display_A_Menu
                       (The_Options_List => Setup_Eop_Options_Menu,
                        Max_Options => 6);
        when 3 => Explain_Where;
                  Option := Display_A_Menu
                       (The_Options_List => Setup_Eop_Options_Menu,
                        Max_Options => 6);
        when 4 => Dump_Critical_Path(Exec_Trace);
                  Option := Display_A_Menu
                       (The_Options_List => Setup_Eop_Options_Menu,
                        Max_Options => 6);
        when 5 => Dump_Execution_Trace(Exec_Trace);
                  Option := Display_A_Menu
                       (The_Options_List => Setup_Eop_Options_Menu,
                        Max_Options => 6);
        when 6 => exit;
        when others => null;
      end case;
    end loop;
  end Access_End_Of_Processing_Ef;
```

```
--| Subroutine Name:  ACCESS_RUNTIME_EF
--| Inputs: None
--| Outputs: None
--| Purpose: Provide access from ES shell to explanation facility.
--| Notes: None

  procedure Access_Runtime_Ef is
    Choice : Positive := Display_A_Menu
                         (The_Options_List => Setup_Access_To_Ef_Menu,
                          Max_Options => 2);
  begin
    case Choice is
      when 1 => Explain_Runtime;
      when others => null;
    end case;
  end Access_Runtime_Ef;

end Ef;  --body
```

Appendix B

Generic STACK Code

This is the generic stack package that is used for the execution trace in Package Ef. It was taken from a library of reusable software components developed by Grady Booch.

```
--|*********************************************************************
--|*********************************************************************
--|*********************************************************************
--|*********                                                   *********
--|*********        GENERIC   STACK   SPECIFICATION            *********
--|*********                                                   *********
--|*********************************************************************
--|*********************************************************************
--|*********************************************************************
generic
        type Item is private;

package Stack_Sequential_Bounded_Managed_Noniterative is

type Stack ( The_Size: Positive ) is limited private;

        -- CONSTRUCTORS

procedure Copy ( From_The_Stack: in Stack;
                 To_The_Stack: in out Stack );

procedure Clear ( The_Stack: in out Stack );

procedure Push ( The_Item: in Item; On_The_Stack: in out Stack );
        -- raise overflow: exception whenever no free space exist
        -- for The_Item in On_The_Stack

procedure Pop ( The_Stack: in out Stack );
        -- raise underflow: exception whenever The_Stack is empty


        -- SELECTORS

function Is_Equal ( Left: in Stack; Right: in Stack ) return Boolean;
```

```
function Depth_Of ( The_Stack: in Stack ) return Natural;

function Is_Empty ( The_Stack: in Stack ) return Boolean;

function Top_Of ( The_Stack: in Stack ) return Item;

        -- EXCEPTION

Overflow: exception;

Underflow: exception;


private

        type Items is array ( Positive range <> ) of Item;

        type Stack ( The_Size: Positive ) is record
                The_Top: Natural := 0;
                The_Items: Items ( 1 .. The_Size );
                end record;

end Stack_Sequential_Bounded_Managed_Noniterative; -- spec
```

```
--|*************************************************************
--|*************************************************************
--|*************************************************************
--|*********                                           *********
--|*********            GENERIC   STACK   BODY         *********
--|*********                                           *********
--|*************************************************************
--|*************************************************************
--|*************************************************************

package body Stack_Sequential_Bounded_Managed_Noniterative is


procedure Copy ( From_The_Stack: in Stack;
                 To_The_Stack: in out Stack ) is
begin
if From_The_Stack.The_Top > To_The_Stack.The_Size then
        raise Overflow;
else
        To_The_Stack.The_Items ( 1 .. From_The_Stack.The_Top ) :=
           From_The_Stack.The_Items ( 1 .. From_The_Stack.The_Top );
        To_The_Stack.The_Top := From_The_Stack.The_Top;
end if;
end Copy;

procedure Clear ( The_Stack: in out Stack ) is
begin
The_Stack.The_Top := 0;
end Clear;


procedure Push ( The_Item: in Item; On_The_Stack: in out Stack ) is
        -- raise overflow: exception whenever no free space exist
        -- for The_Item
in On_The_Stack
begin
On_The_Stack.The_Items ( On_The_Stack.The_Top + 1 ) := The_Item;
On_The_Stack.The_Top := On_The_Stack.The_Top + 1;
exception
        when Constraint_Error =>
                raise Overflow;
end Push;

procedure Pop ( The_Stack: in out Stack ) is
        -- raise underflow: exception whenever The_Stack is empty
begin
The_Stack.The_Top  := Natural'Pred ( The_Stack.The_Top );
exception
        when Constraint_Error =>
                raise Underflow;
end Pop;
```

```
        -- SELECTORS

function Is_Equal ( Left: in Stack; Right: in Stack )
                                        return Boolean is
begin
if Left.The_Top /= Right.The_Top then
        return False;
else
        for Index in 1 .. Left.The_Top loop
          if Left.The_Items ( Index ) /= Right.The_Items (Index) then
                        return False;
                end if;
        end loop;
end if;
end Is_Equal;


function Depth_Of ( The_Stack: in Stack ) return Natural is
begin
return The_Stack.The_Top;
end Depth_Of;

function Is_Empty ( The_Stack: in Stack ) return Boolean is
begin
return ( The_Stack.The_Top = 0 );
end Is_Empty;

function Top_Of ( The_Stack: in Stack ) return Item is
begin
return The_Stack.The_Items ( The_Stack.The_Top );
exception
        when Constraint_Error =>
                raise Underflow;
end Top_Of;

end Stack_Sequential_Bounded_Managed_Noniterative; -- body
```

Appendix C

ES Shell Code

The following code represents those subroutines in the original ES shell code that were modified in order to *plug-in* the explanation facility. Added code is identified by bold faced type.

```
--  ************************************
--  *                                  *
--  *  Fire_Rules                      *  SPEC & BODY
--  *                                  *
--  ************************************

procedure Fire_Rules
  (Frame
     : in Frames.Frame_Ptr;
   Rule_List : in Frames.Rule_List_Ptr) is

--|  Purpose: This procedure controls firing the rules for a demon.
--|
--|  Exceptions:  None.
--|
--|  Notes:  None.
--|

Temp_Rules
  : Frames.Rule_List_Ptr := Rule_List;
Current_Rule
  : Rules.Rule_Ptr;
Temp_Frame
  : Frames.Frame_Ptr := Frame;

begin
  while Temp_Rules /= null loop
    -- Find the rule pointer since only the number is kept in demon
    Current_Rule := Rules.Find_Rule (Temp_Rules.Rule_No);
    -- Incrament the rules tested count
    Frames.No_Rules_Tested := Frames.No_Rules_Tested + 1;
```

```
      Ef.Build_Execution_Trace(Temp_Frame, Current_Rule);


    if Current_Rule.Inactive and then
             Test_Antecedent (Frame, Current_Rule.Antecedent) then
      -- Incrament the rules fired count
      Frames.No_Rules_Fired := Frames.No_Rules_Fired + 1;
      User_Io.Print_Rule_No (Current_Rule.Rule_No);
      Text_Io.Put_Line (" fired successfully.");


      Ef.Set_Rule_Fired(Current_Rule);



      Current_Rule.Inactive := False;
      Fire_Consequent (Frame, Current_Rule.Consequent);
      Current_Rule.Inactive := True;
      Temp_Rules := null;
    else
      User_Io.Print_Rule_No (Current_Rule.Rule_No);
      Text_Io.Put_Line (" not fired. ");


      Ef.Set_Rule_Failed(Current_Rule);



      Temp_Rules := Temp_Rules.Next;
    end if;
  end loop;
end Fire_Rules;




--  ************************************
--  *                                  *
--  *  Get_User_Input                  *  BODY
--  *                                  *
--  ************************************

function Get_User_Input
  (User_Prompt
     : in Rules.Prompt_Token_Ptr)
  return Frames.Value_Ptr is

--|  Purpose:  Manage user prompts by asking the desired questions
--|            and returning the response in the value form.
--|
--|  Exceptions:  None.
--|
```

```
--|  Notes:  None.

Result
  : Frames.Value_Ptr;

begin
  Text_Io.New_Line;
```

Print_String (User_Prompt.Prompt_Msg);
Ef.Access_Runtime_Ef;

```
  Print_String (User_Prompt.Prompt_Msg);
  Clear;       -- clear the input buffer
  Result := new Frames.Value (User_Prompt.Prompt_Value.V_Type);
  Result.V_Type := User_Prompt.Prompt_Value.V_Type;
  Read_Value (Result);
  return Result:
end Get_User_Input;




--  ************************************
--  *                                  *
--  *  Print_Menu                      *  BODY
--  *                                  *
--  ************************************

procedure Print_Menu
  (Knowledge_Base_Loaded
     : in Boolean) is

--|  Purpose:  Rountine used to print the user menu to operate shell.
--|
--|  Exceptions:  None.
--|
--|  Notes:  None.
--|

begin
  Text_Io.New_Line (25);
  Text_Io.Set_Col (12);
  Text_Io.Put("EXPERT SYSTEM SHELL MAIN MENU");
  Text_Io.New_Line (2);
  Text_Io.Set_Col (8);
  Text_Io.Put("1)  Load and Run Knowledge Base.");
  if Knowledge_Base_Loaded then
    Text_Io.New_Line (2);
    Text_Io.Set_Col (8);
```

```
    Text_Io.Put("2)  Print Frames to a disk file.");
    Text_Io.New_Line (2);
    Text_Io.Set_Col (8);
    Text_Io.Put("3)  Print Rules to a disk file.");
    Text_Io.New_Line (2);
    Text_Io.Set_Col (8);
    Text_Io.Put("4)  Print Statistics to screen.");
    Text_Io.New_Line (2);
    Text_Io.Set_Col (8);
    Text_Io.Put("5)  Print Single Frame to Screen");
    Text_Io.New_Line (2);
    Text_Io.Set_Col (8);
    Text_Io.Put("6)  Reload the Frames and Rerun");
    Text_Io.New_Line (2);
    Text_Io.Set_Col (8);
    Text_Io.Put("7)  Exit Shell");

Text_Io.New_Line (2);
Text_Io.Set_Col(8);
Text_Io.Put("8) Access End-Of-Processing EF");


  end if;
end Print_Menu;




--  ************************************
--  *                                  *
--  *  Shell                           *  SPEC & BODY
--  *                                  *
--  ************************************

with Text_Io,
     Frames,
     Rules,
     User_Io,
     K_B_File,
     Ef,
     Knowledge_Base;


procedure Shell is

--|  Purpose:  Calling routine to begin execution of the shell
--|
--|  Exceptions:  None.
--|
--|  Notes:
```

```
--|

function "=" (X, Y : in Frames.Frame_Ptr) return Boolean
  renames Frames."=";

Choice
  : Character;
Desired_Frame
  : Frames.Frame_Name;
Desired_Frame_Ptr
  : Frames.Frame_Ptr;
Knowledge_Base_Loaded
  : Boolean := False;
The_Name
  : String (1..80);
The_Size
  : Natural;

begin
  loop
    User_Io.Print_Menu (Knowledge_Base_Loaded);
    Text_Io.New_Line (2);
    Text_Io.Set_Col (4);
    Text_Io.Put ("Enter Choice => ");
    Text_Io.Get (Choice);
    if not Knowledge_Base_Loaded and Choice /= '1' and
       Choice /= '7' then
      Choice := '0';
    end if;

    -- clear input stream
    Text_Io.Skip_Line;

    case Choice is
      when '0' =>
           Text_Io.Put_Line ("No Knowledge Base Loaded in shell.");
      when '1' =>

            Ef.Clear_Execution_Trace;


           Knowledge_Base.Build_Knowledge_Base;
           Knowledge_Base_Loaded := True;
           Text_Io.Skip_Line;
      when '2' =>
           Text_Io.Put_Line ("Open an output file.");
           Text_Io.Put_Line ("Printing the frames ... ");
           User_Io.Get_File_Name (The_Name, The_Size);
           K_B_File.Open_Output (The_Name, The_Size);
           User_Io.Print_Active_Frames;
           K_B_File.Close_Output;
```

```
            Text_Io.Put_Line ("Output file closed.");
        when '3' =>
            Text_Io.Put_Line ("Open an output file.");
            Text_Io.Put_Line ("Printing the rules ... ");
            User_Io.Get_File_Name (The_Name, The_Size);
            K_B_File.Open_Output (The_Name, The_Size);
            Text_Io.Put_Line ("Listing of the active rules.");
            Text_Io.New_Line (2);
            User_Io.Print_Rules(Rules.Active_Rules);
            K_B_File.Close_Output;
            Text_Io.Put_Line ("Output file closed.");
        when '4' =>
            User_Io.Print_Stats (Frames.No_Rules_Tested,
                                 Frames.No_Rules_Fired);
        when '5' =>
            Text_Io.Put ("Enter the frame name => ");
            Desired_Frame := User_Io.Read_Frame_Name;
            Desired_Frame_Ptr := Frames.Find_Frame (Desired_Frame);
            if Desired_Frame_Ptr = null then
            Text_Io.Put_Line
              ("Selected Frame Name is not on the active list");
            else
               User_Io.Print_Frame_Contents (Desired_Frame_Ptr);
            end if;
            Text_Io.Skip_Line;
        when '6' =>

              Ef.Clear_Execution_Trace;


            Knowledge_Base.Reload_Frames;
            Text_Io.Skip_Line;
        when '7' =>
            exit;

        when '8' =>
Ef.Access_End_Of_Processing_Ef;


        when others =>
            Text_Io.Put_Line ("Bad Menu number entered.");
    end case;

    -- Delay for readibility
    Text_Io.New_Line(2);
    Text_Io.Put_Line ("Return to continue...");
    Text_Io.Get_Line (The_Name, The_Size);
  end loop;
end Shell;
```

Bibliography

[1] G. Booch. *Software Engineering with Ada.* The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, second edition, 1983.

[2] B. Buchanan and E. Shortliffe. *Rule-Based Expert Systems.* Addison-Wesley Publishing Co., Inc., Menlo Park, California, 1984.

[3] C. W. Butler, E. D. Hodil, and G. L. Richardson. Building knowledge-based systems with procedural languages. *IEEE Expert Magazine*, pages 47–59, Summer 1988.

[4] T. Bylander and B. Chandrasekaran. Generic tasks for knowledge-based reasoning: The 'right' level of abstraction for knowledge acquisition. Technical Research Report 87-TB-KNOWAC, The Ohio State University Department of Computer and Information Science Laboratory for Artificial Intelligence Research, 1987.

[5] J. C. Cardow. Toward an expert system shell for a common ada programming support environnment. Master's thesis, Wright State University, Dayton, Ohio, 1989.

[6] B. Chandrasekaran. Generic tasks in knowledge-based reasoning: High level building blocks for expert system design. Technical Research Report 86-BC-IEEEX, The Ohio State University Department of Computer and Information Science Laboratory for Artificial Intelligence Research, 1986.

[7] B. Chandrasekaran. Towards a functional architecture for intelligence based on generic information processing tasks. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1183–1192, 1987.

[8] B. Chandrasekaran, J. Josephson, and M. C. Tanner. Explaining control strategies in problem solving. *IEEE Expert Magazine*, pages 9–24, Spring 1989.

[9] B. Chandrasekaran, J. R. Josephson, and A. Keuneke. Functional representation as a basis for generating explanations. Technical Research Report 86-BC-FUNEXPL, The Ohio State University Department of Computer and Information Science Laboratory for Artificial Intelligence Research, 1986.

[10] W. J. Clancy. The epistemology of a rule-based expert system - a framework for explanation. *ARTIFICIAL INTELLIGENCE An International Journal*, 20(3):215–251, May 1983.

[11] M. Coombs, editor. *Developments In Expert Systems.* Academic Press, Inc., London, England, 1984.

[12] R. Davis. Applications of meta level knowlege to the construction, maintenance and use of large knowledge bases. *SAIL AIM-283*, 1976.

[13] R. Davis. Meta-rules: Reasoning about control. *ARTIFICIAL INTELLIGENCE An International Journal*, 15(3):179–222, Dec 1980.

[14] R. Davis and D. Lenat. *Knowledge-Based Systems In Artificial Intelligence*. McGraw-Hill, New York, New York, 1982.

[15] E. A. Feigenbaum, B. G. Buchanan, and J. Lederberg. On generality and problem solving: A case study using the dendral program. In *Machine Intelligence*, pages 165–190. Edinburgh University Press, 1971.

[16] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, Sep 1985.

[17] M. W. Firebaugh. *ARTIFICIAL INTELLIGENCE: A Knowledge-Based Approach*. Boyd and Fraser Publishing Co., Boston, Massachusetts, 1988.

[18] R. Forsyth, editor. *Expert Systems: Principles and Case Studies*. Chapman and Hall Publishing Co., London, England, 1984.

[19] K. J. Hammond. Explaining and repairing plans that fail. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 109–114, 1987.

[20] J. R. Josephson et al. A mechanism for forming composite explanatory hypotheses. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 445–454, May/June 1987.

[21] J. R. Josephson, J. Smith, and B. Chandrasekaran. Assembling the best explanation. Technical Research Report 84-JJ-RED, The Ohio State University Department of Computer and Information Science Laboratory for Artificial Intelligence Research, 1984.

[22] M. Minsky. A framework for representing knowledge. In *The Psychology of Human Vision*, chapter 6. McGraw-Hill Publishing, 1975.

[23] S. Mittal and B. Chandrasekaran. Deep versus compiled approaches to diagnostic problem-solving. Technical Research Report 83-BC-DEEPCOM, The Ohio State University Department of Computer and Information Science Laboratory for Artificial Intelligence Research, 1983.

[24] J. D. Moore, R. Neches, and W. R. Swartout. Enhanced maintenance and explanation of expert systems through explicit models of their development. *IEEE Transactions on Software Engineering*, SE-11(11):1337–1350, Nov 1985.

[25] H. E. Pople. The formation of composite hypotheses in diagnostic problem solving. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1030–1037, 1977.

[26] R. Rubinoff. Explaining concepts in expert systems: The clear system. In *Proceedings of the Second Conference on Aritificial Intelligence Applications*, pages 416–421, 1985.

[27] R. C. Schank. *Explanation Patterns: Understanding Mechanically and Creatively.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1986.

[28] A. C. Scott, W. J. Clancy, R. Davis, and E. H. Shortliffe. Methods for generating explanations. In *Rule-Based Expert Systems*, chapter 18. Addison-Wesley Publishing Co., Inc., 1971.

[29] V. Sembugamoorthy and B. Chandrasekaran. Functional representation of devices and compilation of diagnostic problem-solving systems. Technical Research Report 86-VS-FUNCT, The Ohio State University Department of Computer and Information Science Laboratory for Artificial Intelligence Research, 1986.

[30] E. M. Shortliffe. *MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection.* PhD thesis, Stanford University, 1974.

[31] J. R. Slagle and M. R. Wick. An explanation facility for today's expert systems. *IEEE Expert Magazine*, pages 26–36, Spring 1989.

[32] S. W. Smoliar and W. R. Swartout. On making expert systems more like experts. In *AI Tools and Techniques*, pages 197–216. Ablex Publishing Corporation, 1989.

[33] J. Stein, editor. *The Random House College Dictionary.* Random House, Inc., New York, New York, revised edition, 1979.

[34] W. R. Swartout. A digitalis therapy advisor with explanations. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 819–825, 1977.

[35] W. R. Swartout. Xplain: A system for creating and explaining expert consulting programs. *ARTIFICIAL INTELLIGENCE An International Journal*, 21(3):285–325, Sep 1983.

[36] D. Warner Hasling, W. J. Clancy, and G. Rennels. Strategic explanations for a diagnostic consultation system. *International Journal Man-Machine Studies*, pages 3–19, Jan 1984.

[37] J. L. Weiner. Blah: A system which explains its reasoning. *ARTIFICIAL INTELLIGENCE An International Journal*, 15(1,2):19–48, Nov 1980.